Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

2005-03

API development for persistent data sessions support

Pailom, Chayutra

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/2223

Copyright is reserved by the copyright owner

Downloaded from NPS Archive: Calhoun

DUDLEY KNOX LIBRARY

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

API DEVELOPMENT
FOR PERSISTENT DATA SESSIONS SUPPORT

by

Chayutra Pailom

March 2005

Thesis Advisor: Su Wen
Thesis Co-Advisor: Arijit Das

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis
4. TITLE AND SUBTITLE: API Development for Persistent Data Sessions Support		**5. FUNDING NUMBERS**
6. AUTHOR(S) Chayutra Pailom		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis studies and discusses the development of the API, called the persistency API, for supporting the persistent data sessions. Without persistent session support, network applications often need to be restarted from the beginning when intermittent physical connection loss happens. Application programmers can use the persistency API to achieve the service continuity. The persistency API provides the interface that allows a program to continue retrieve data from the point the connection is lost after the physical connection is restored. The focus of this thesis is to develop a generalized persistency API that supports various types of applications. This thesis studies the persistent session support for two types of transport protocols, TCP and UDP, which are used by major network applications. An application that performs text file and video file transfer is implemented to demonstrate the persistent data transfer sessions for TCP and UDP, respectively. The study shows that the proposed APIs can support the data transfer continuity in the reconnection process.

14. SUBJECT TERMS Persistency API, M-TCP, UDP, TCP, PFTP	**15. NUMBER OF PAGES** 165
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release, distribution is unlimited

API DEVELOPMENT FOR PERSISTENT DATA SESSIONS SUPPORT

Chayutra Pailom
Captain, Royal Thai Army
B.S., Chulachomklao Royal Military Academy, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2005

Author: Chayutra Pailom

Approved by: Su Wen
Thesis Advisor

Arijit Das
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis studies and discusses the development of the API, called the persistency API, for supporting the persistent data sessions. Without persistent session support, network applications often need to be restarted from the beginning when intermittent physical connection loss happens. Application programmers can use the persistency API to achieve the service continuity. The persistency API provides the interface that allows a program to continue retrieve data from the point the connection is lost after the physical connection is restored. The focus of this thesis is to develop a generalized persistency API that supports various types of applications. This thesis studies the persistent session support for two types of transport protocols, TCP and UDP, which are used by major network applications. An application that performs text file and video file transfer is implemented to demonstrate the persistent data transfer sessions for TCP and UDP, respectively. The study shows that the proposed APIs can support the data transfer continuity in the reconnection process.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to dedicate this thesis, the result of two years and three months graduate study at the Naval Postgraduate School, to my family who has encouraged me through the process.

I would like to express my appreciation to my advisor, Professor Su Wen. I could not have imagined having a better supervisor and mentor for my master, and without her commonsense, knowledge, perceptiveness, supervision and support I would never have finished. I would also like to thank my Co-Advisor, Arijit Das, for his administrative and technical assistance and encouragement to read the whole thing so thoroughly.

Finally, I want to thank my country and Royal Thai Supreme Command who gave the opportunity to have this Master's degree in Computer Science in the United States of America.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

One of the problems during the Signal operation in Thailand is the lack of service continuity of the data communication between end users at the application level. To complete the mission, the applications need to be fully transferred regardless if the state of the physical connection is disrupted or not. Currently, an error or the loss of the physical connection during the data transfer sessions can cause the disruption in the application operation. It requires manual restarting of the application. If the physical connection is lost either from the system itself or from the interference, it is not easy to reconnect at the beginning of the session and retransmit the entire data in order to provide the reliable content delivery. However, in each Signal operation, such applications have to be resubmitted as soon as possible to ensure the survivability of the service. To achieve the goal of the mission for this case, not only must the service survivability be established but the applications also need to operate seamlessly in the face of a physical connection loss. The interrupt data or the state of the connection must be continued from the point it was stopped.

B. SCOPE AND METHODOLOGY

This thesis develops an application-level support of persistent sessions to various applications. Specifically, APIs for general applications using persistent connection are designed and implemented.

The research follows the methodology that can be conducted as follows. Firstly, the research starts with searching for the existing protocol emphasizing File Transfer Protocol, Real Time Streaming Protocol, and Telnet. The study of these protocols can be background knowledge in order to extend this concept for this research. The research then goes on specifying each protocol parameters and application requirements for both user interface and the connection protocols. This area has to be studied in depth because the result from development for each parameter effects the system. The critical session of this research is to design and develop a reconnection session using API for each application protocol. The development of the API can be done using the supporting idea above and this API can be used for generic applications in the future. Finally, the testing phase for the implemented API needs to be achieved in order to guarantee the service for persistent sessions. This testing is conducted on a wired environment to reconnect the FTP, RTSP or Telnet respectively. If time permits, the migration of each protocol will be developed to support this research.

Since Java programming language is popularly used in computer network programming and provides a good modularity, it was chosen to develop the APIs for this thesis. Currently, there are a number of classes in Java that can be used to support the reconnection session for network protocols. However, each protocol has to be written by the programmers on their own to provide the reconnection session when it lost. No APIs specific for persistent session support are currently available for programmers to use directly. This thesis focuses on

building new APIs that programmers can write applications with regardless of the underlying network protocols the applications are using.

The network environment used for this thesis work was based on a wired environment. The research was performed on an end-to-end user's connection. The implementation was conducted using a connection on the same network and used a variety of protocols to transmit the information and test the API. A wireless scenario was an additional environment tested for mobile device. As a result, this API can be extended in its capability to fully support the wireless environment. Therefore, both client and server for this research was a workstation running a Windows operating system.

The thesis did not consider the following as part of the implementation:

- *Security*: The security aspect of the development is not covered. Even though it will be used for the military operations, the initial purpose for this development is to achieve the survivability of the service. There is a security tunneling design that would be appropriate to encapsulate for each session. Future research may be necessary for the extension of this work to increase the level of security.
- *Scalability*: Even though the multitasking approach is suitable for designing on the server-side application transferring the data to other hosts, this thesis does not address the capability to support a large number of users that produce heavy traffic as a single server.
- *Speed of recovery*: For this thesis work, speed is not an important factor of an experiment. The speed of the

transmission is forced by the physical connection media and performance of both the sender and receiver, which are not the main factors to be observed.

C. RESEARCH QUESTIONS

The following questions are considered in this thesis:

- What are the key components of persistent session service?
- What are the key components that must be implemented in preparing API to support application layer needs?
- How can this API support the reconnection session?
- What types of applications can benefit most from the protocol developed?
- How can this communication supplement be flexible when the availability of the information server varies?

D. ORGANIZATION

The covered material is organized into the following chapters in order to fulfill the objectives of this thesis. Chapter II covers the background and related works that provide the introduction to Persistent Data Sessions and previous research works. Chapter III refers to the on-going Reliable Content Delivery using Persistent Data Sessions along with the development of Application Programming Interface related to this thesis project. Chapter IV covers the design and implementation of the thesis process. It describes the software design and programming procedure of the prototype in detail. It also mentions how the most important features of the API are supported in the reconnection state. Chapter V summarizes the testing phases of the API development and objects models. Specific

scenarios are conducted for each protocol appropriately. Finally, Chapter VI presents the conclusion, recommendation and the future work to be continued in the establishment of persistent data sessions by extending this thesis work to support the wireless environment.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND RELATED WORKS

This chapter will discuss the general knowledge and the previous research regarding persistent data sessions that are related to the concept of this thesis work. The concept of persistent data sessions and an overview of the existing network-continuity protocols that have the reconnection characteristics will be introduced separately.

A. RELIABLE CONTENT DELIVERY WITH PERSISTENT DATA SESSIONS

1. Background

This section refers to the general idea of persistent data sessions which provides reliable content delivery of the data communication. According to TCP/IP model, this procedure is referred to the process at the application layer. Before going into the detail of persistent sessions, the term ‘session’ will be described as a fundamental idea for this section. A session is either a lasting connection using the session layer of a network protocol or a lasting connection between a user (or user agent) and a peer, typically a server, usually involving the exchange of many packets between the user's computer and the server. A session is typically implemented as a layer in a network protocol [1]. In the case of transport protocols which do not implement a formal session layer or where sessions at the session layer are generally very short-lived, sessions are maintained by a higher level program using a method defined in the data being exchanged.

To get the information from a server, a system may issue a “request” packet to the server. If a “reply” packet

arrives in response, the transferring session will be started by using one of the following transport protocols:

- **Transmission Control Protocol (TCP)** providing a one-to-one connection oriented, reliable communication service of the sequence and acknowledgement of packets sent and recovery of packets lost during transmission.
- **User Datagram Protocol (UDP)** providing one-to-one or one-to-many connectionless, unreliable connection service which is used when the amount of data to be transferred is small, when the overhead of establishing a TCP connection is not desired, or when the application or upper-layer protocol provide reliable delivery.

Extended from the concept of the term 'session', persistent data sessions, or persistent connections which are sometimes called "keep-alive" connections or "connection reuse," can be used to optimize the way servers return content to the client. It is the idea of using the transport protocol connection to send and receive multiple requests or responses, as opposed to opening a new one for every single request or response pair. As proposed, the client can send multiple requests on a single connection. This capability is negotiated in response to the first request on a connection. The server can choose how many requests it will allow on a persistent connection and also how long to wait for subsequent requests before terminating the connection. Most servers will allow you to configure these things.

There are several advantages of using persistent connections [2] including:

- Network friendly. Less network traffic due to fewer setting up and tearing down of connections.
- Reduced latency on subsequent request. Due to avoidance of initial protocol handshake
- Long lasting connections allowing protocols sufficient time to determine the congestion state of the network, thus to react appropriately.

B. EXISTING APPROACHES IN PROVIDING PERSISTENT DATA SESSIONS

This section is about the existing protocols related to the service continuity characteristic. The concept of both kinds of protocol is related to each other but different in the level of operation. M-TCP is resided in the transport layer which provides the guarantee service similar to TCP but has additional features in order to achieve the service continuity. Another protocol called PFTP is a little bit different from the first protocol because it is developed at the application layer which uses TCP as an underlying protocol. The details of each will be described following.

1. Migratory Transmission Control Protocol (M-TCP)

a. Overview

This protocol is proposed from the Laboratory for Network Centric Computing (Disco Lab) at Rutgers University [3]. The concept of this protocol is called service continuity. M-TCP is a transport layer protocol for building highly available network services by means of transparent migration of the server endpoint of a live connection between cooperating servers that provide the same service. The origin and destination server hosts cooperate by transferring supporting state in order to accommodate the migrating connection. It is one of the

network reconnection solutions to reestablish another connection using the point that lost the connection to the previous server to try to reconnect to another server for retrieving the same information at the point it was lost. Therefore, the two servers that serve such clients have to have good coordination between each other.

The client starts a service session by connecting to a preferred server, which supplies the addresses of its cooperating servers, along with authentication information. The client endpoint of a connection can initiate a migration by contacting one of the alternate servers. The migration trigger may reside with the client or with any of the servers. The server endpoint of the connection migrates between the cooperating servers, transparent to the client application.

This protocol is compatible with TCP in which the client protocol stack can initiate migration of the remote end point of live connection to an alternate server. Migration is transparent to the client application. M-TCP decouples the migration mechanism from migration policy that specifies when a connection should migrate. Migration may be triggered according to some migration policy under conditions like server overload network congestion, the loss of physical connection, degradation in performance perceived by client, etc.

b. Goals and Features of M-TCP

The goal of the M-TCP is to support the efficiency of live connections. It also offers a better alternative than the simple retransmission to the same server, which may be suffering from overload or Denial of Service attack, or may be known or maybe not be easily

reachable due to congestion, and decouples a given service from the unique/fixed identity of its provider.

This protocol has features as the following. It is general and flexible which means that it doesn't rely on knowledge about a given server application or application level protocol. It allows fine-ground migration of live individual connection, unlike heavyweight process migration schemes, and it is symmetric with respect to and decoupled from any migration policy

c. M-TCP Mechanism

The M-TCP design assumes that the state of the server application can be logically split among connections by defining a fine-grained state associated with each connection.

The M-TCP service interface can be best described as a contract between the server application and the transport protocol. According to this contract, the application must execute the following actions: *(i) export* a state snapshot at the old server, when it is consistent with data sent/received on the connection; *(ii) import* the last state snapshot at the new server after migration, to resume service to client. In exchange, the protocol: *(i) transfers* the per-connection state to the new server and *(ii) synchronizes* the per-connection application state with the protocol state.

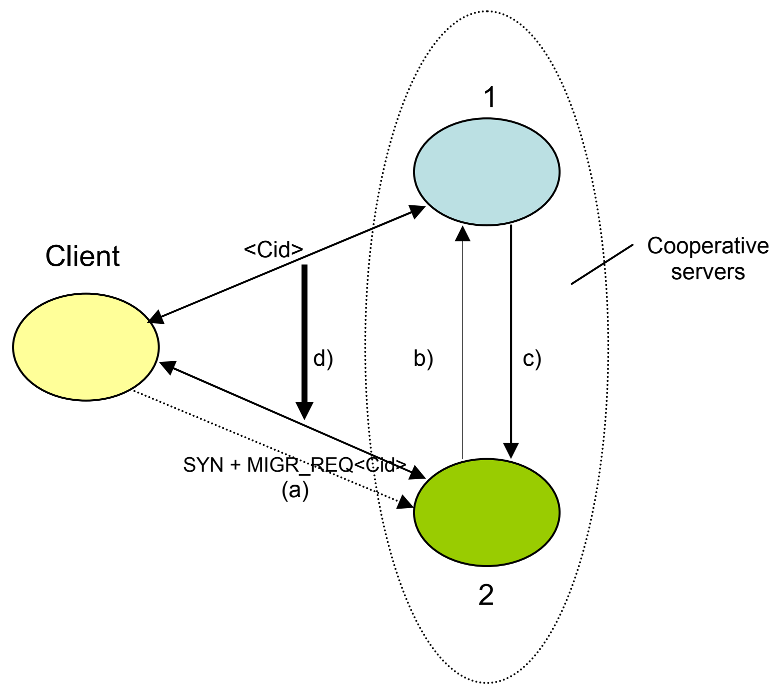

Figure 1. Migration mechanism in M-TCP.

From Figure 1, Connection Cid, initially established by client C with server S2, migrates to alternate server S2. The migration mechanism of M-TCP ensures that the new server resumes service while preserving the exactly-once delivery semantics across migration, without freezing or otherwise disrupting the traffic on the connection. The client application does not need to change.

A client contacts the service through a connection Cid to a preferred server S1. At the connection setup, S1 supplies the addresses of its cooperating servers, along with migration certificates. The client-side M-TCP initiates migration of Cid by opening a new connection to an alternate server S2, sending the migration certificate in a special option. (Figure 1(a)). To

reincarnate Cid at S2, M-TCP transfers associated state (protocol state and the last snapshot) from S1.

Depending on the implementation, the state transfer can be either *(i)* lazy (on-demand), i.e., it occurs at the time migration is initiated, or *(ii)* eager, i.e., it occurs in anticipation of migration, e.g., when a new snapshot is taken. Figure 1 shows the lazy transfer version: S2 sends a request (b) to S1 and receives the state (c). If the migrating endpoint is reinstated successfully at S2, then C and S2 complete the handshake, which ends the migration (d).

Upon accepting the migrated connection, the server application at S2 imports the state snapshot. It then resumes service using the snapshot as a restart point, and performs execution replay for a log-based recovery supported by the protocol. The execution replay restores the state of the service at the new server and synchronizes it with the protocol state. To support the replay, M-TCP logs and transfers from S1 data received and acknowledged since the last snapshot. It also transfers unacknowledged data sent before the last snapshot, for retransmission from S2.

2. Reliable Content Delivery Using Persistent Data Sessions in Highly Mobile Environment

The work was to develop a client–server file transfer application named *Partial File Transfer Protocol* (PFTP) to demonstrate a possible solution to the problem of a lack of persistent data sessions in wireless mobile networks and LANs. This protocol was developed at the application level which used TCP as an underlying protocol. For this purpose, a prototype communication protocol between the client and the server were designed using Java Technology to achieve

dynamic partial file retrieval in the event of connection loss. The goal is to produce an application user interface that visualizes the partial file retrieval process in real time. This is a proof of concept for service continuity protocol development. Since it has a limitation in transport protocol development, it can support some applications that use TCP as an underlying protocol only. As a result, the concept of this work is to be extended in order to support various applications that use either TCP or UDP as an underlying protocol.

a. PFTP Mechanism

PFTP is an application layer protocol that is based on a client-server communication scheme. For the file transfer process, an application layer communication protocol must be established. The TCP protocol is the underlying protocol for every connection and data transfer between the client and server (Figure 2).

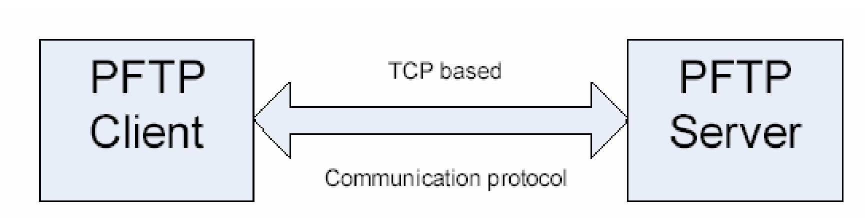

Figure 2. The communication scheme of PFTP application

File Request Message

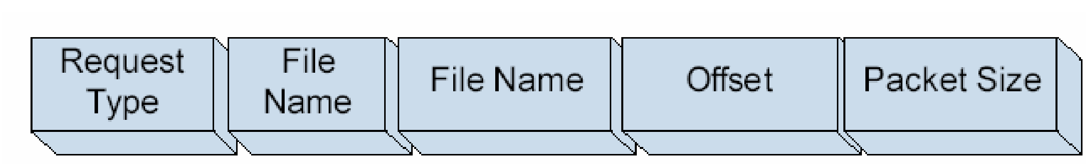

Figure 3. The message request of PFTP application

This communication protocol is shown in Figure 4 [4] and starts with the PFTP server running and waiting for clients at port 6789. When the user opens the PFTP client application, it must choose a PFTP server manually or from a list of already existing servers in order to retrieve the list of files available for transfer. In the predefined server case, an Auto server mode option exists which can be selected when the user wants the application to choose a predefined PFTP server randomly during either the initial available file retrieval or the partial file retrieval process after a connection loss. When the file list is retrieved, a user selects a file name and the packet size and forms a file request packet that is sent to the server. The request packet fields are shown in Figure 3.

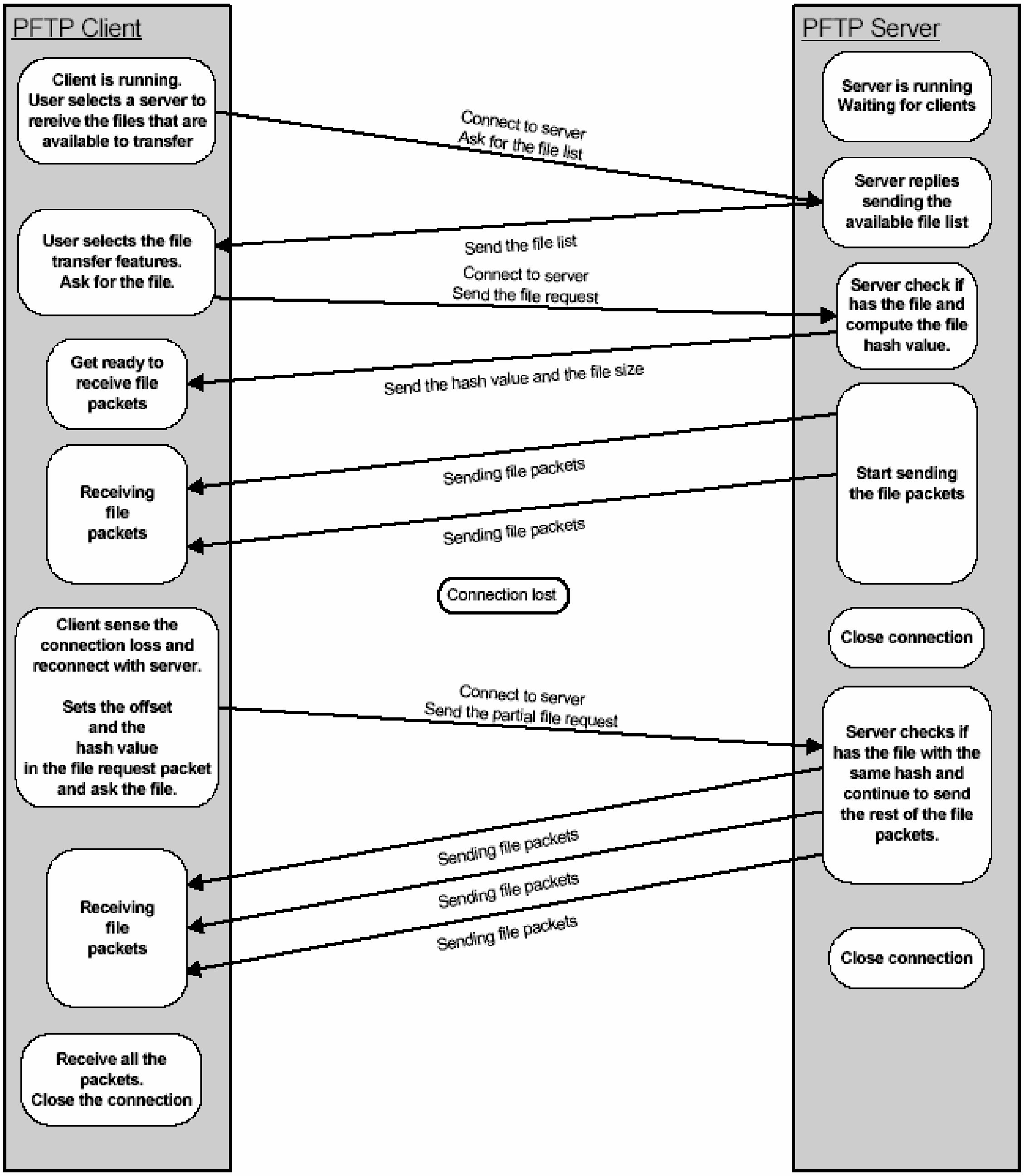

Figure 4. The communication protocol of PFTP application

The server checks if it has the file and, if it does, sends the file size and the hash value of the file back to client. Then the server starts reading the file data and sends it in packets to the client. If, during the

packet transmission, a connection failure occurs, the client side generates partial-file-request packet, setting the offset field value with the next expected packet counter and the hash field value with the file hash received at the start of the file transfer. Then the client attempts to reconnect with the same server or, if the Auto server mode is selected, with the next available PFTP server, and when it is connected, sends the prepared partial-file-request packet.

The server that receives the packet, if the offset value is greater that zero, checks if it has the same file with the same hash value and continues sending the remaining data in packets to the client. Each file connection loss causes the same partial transfer retrieval scheme until the client receives all the file packets and both the client- and server-sides close the connection.

THIS PAGE INTENTIONALLY LEFT BLANK

III. API DESIGNS FOR PERSISTENT DATA SESSIONS

This chapter provides an introduction to the background, capability, and purposes of the APIs used for this thesis. APIs developed for this thesis to support persistent sessions for various applications will be presented.

A. INTRODUCTION TO APPLICATION PROGRAMMING INTERFACE

1. Introduction

An Application Programming Interface (API) defines how programmers utilize particular computers features. Some often used APIs provide programmers with access to display system, file systems, database systems, and networking systems. The APIs developed in this thesis are designed to support a structured approach to network programming. Special attention has been paid to the needs of multimedia applications and to the future requirements of network protocols. After surveying the current approaches, the need was observed for an interface that provides ease of use, extendibility and portability. An object-oriented method that will meet these needs is chosen.

Currently, application programmers are writing more details for underlying functionality and using specific code to interface with the network and transport layers defined by the OSI model. This requires that programmers learn how to communicate with the underlying network. Technicalities include opening and closing communication channels and manipulating data structures. An example of a low level interface used by application programmers is the Berkeley Sockets interface which UNIX systems support [5].

Many applications that run over networks contain this type of interface or one of equivalent complexity. Moreover, the complexity is coupled with redundancy; not only must an interface be written for every application, if a programmer changes the protocol an application is using, the program’s interface to the transport and network layers needs to be rewritten.

This non-portability of applications between network and transport layer protocols could be aggravated by the availability of new, more intricate network level protocols which provide features for multimedia and real-time applications. These protocols are designed to support resource allocation. They allow applications to specify their performance requirements and receive performance guarantees. Although the demand for these services has been firmly established, easy access to the protocols that supply them has not yet been made widely available. With such a variety of protocols and the complexity inherent in implementing them, it would be efficient to supply application programmers with a high-level interface to the underlying layers. The need for improvements in system software in order to not only support the basic reliable protocol but also support real-time and multimedia application has been recognized by previous research. For these reasons, a generic ***Application Programmer Interface (API)*** which acts as a level of abstraction between the application programmer and the network and transport layer protocols are developed. Figure 5 shows the architecture where API abstractly resides within the OSI model.

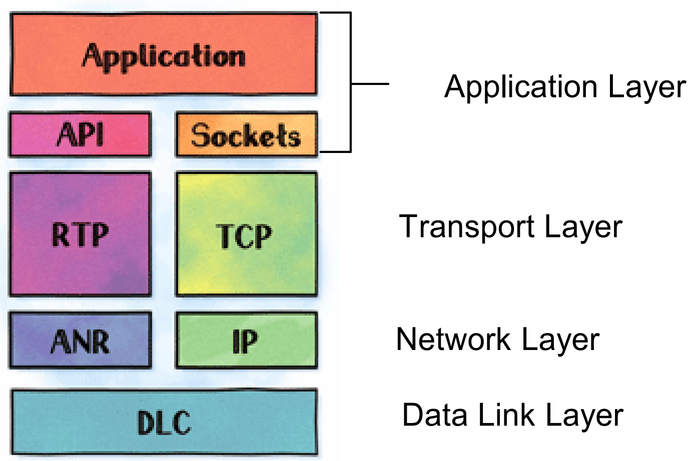

Figure 5. OSI Model with API

The following figure is an example of the data flow from side to side using API support starting from the application layer.

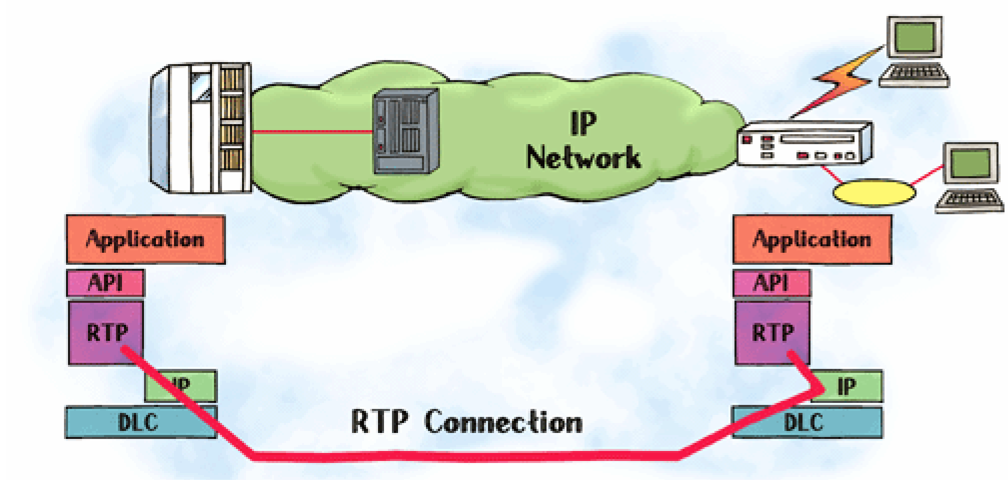

Figure 6. API support for end-to-end connection

B. API SUPPORT FOR PERSISTENT DATA SESSIONS

1. Practical Consideration

During a normal network session, servers will usually have some time-out value beyond which they will no longer maintain an inactive or lost connection. Proxy servers

might make this a higher value since it is likely that the client will be making more connections through the same server. The use of persistent connections places no requirements on the length (or existence) of this time-out for either the client or the server.

When a client or server wishes to time-out, it should issue a graceful close on the transport connection. Clients and servers should both constantly watch for the other side of the transport close, and respond to it appropriately. If a client or server does not detect the other side's close promptly it could cause unnecessary resource drain on the network.

A client, server, or proxy may close the transport connection at any time. For example, a client might have started to send a new request at the same time that the server has decided to close the “idle" or “lost” connection. From the server's point of view the connection is being closed while it was idle, ***but from the client's point of view a request is in progress.***

This means that clients, servers, and proxies must be able to recover from asynchronous close events. Client software should reopen the transport connection and retransmit the aborted sequence of requests without user interaction so long as the request sequence is idempotent. We propose to develop software procedures at the application layer and generate APIs for general use. Non-idempotent methods or sequences must not be automatically retried, although user agents may offer a human operator the choice of retrying the request(s). Confirmation by user-agent software with semantic understanding of the application may substitute for user confirmation.

Servers should always respond to at least one request per connection, if at all possible. Servers should not close a connection in the middle of transmitting a response, unless a network or client failure is suspected.

2. Problem Concern

From a consideration above, the problem will be mainly focused on the lost state. When a physical connection is lost, that means the connection has to be reestablished and the data has to be retransmitted from the beginning of the file. For TCP connection, even though it uses a connection-oriented mechanism which controls the content of the data to be sent appropriately, it is necessary to make a new connection manually for this scenario. The API will help the programmers during this session by automatically detecting the physical connection and making a new connection instead of the client. In case of an unreliable connection like UDP, the data is being sent regardless of the packet loss. To be a persistent session, the proposed API will control the rest of the data in order to achieve the virtual reliable protocol as a persistent data session.

In order to assist users in programming the process of reconnection, the procedure of the reconnection phase will have to be provided step-by-step. There is no programming library that can support all of the steps for the reconnection process. This is also applied to the states of connectionless session. The lost connection has to be reestablished in order to complete sending the file to the destination.

As above, the API is another proposed solution for an easier reconnection. Such a package can be called to manipulate the physical connection instead of doing so manually. All the programmers need is to call this package

which consists of the related functions to reconnect the ongoing session when the physical connection has been lost. Therefore, while the data is being transferred to the destination, the state of the connection and the critical parameters the program is using at that time have to be tracked or recorded in order to make a reconnection without prompting the user. The following figure is an example of the API process for mobility client [6]. This client can be both mobility and stationary object.

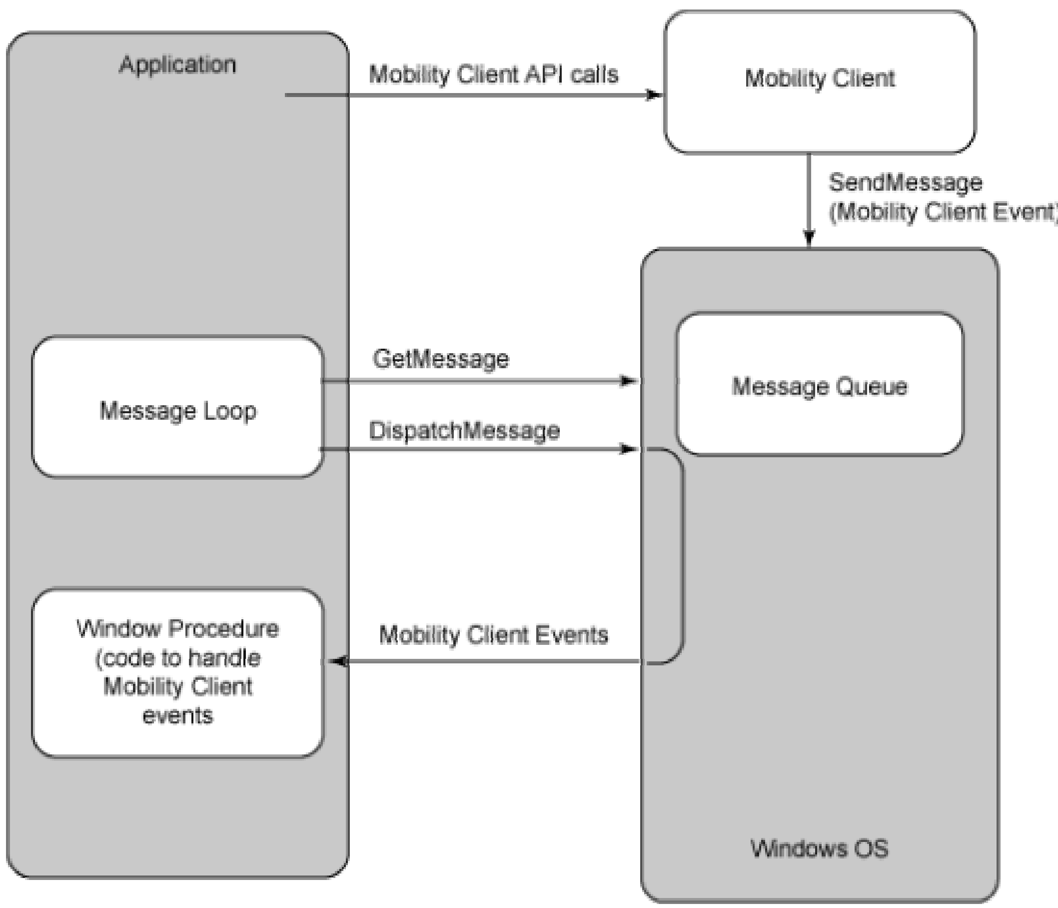

Figure 7. API message flows

C. API DESIGN FOR VARIOUS NETWORK APPLICATIONS

1. Purpose

For this thesis project, the APIs are not developed for specific applications, but aim to be used by a multitude of programs that use either TCP or UDP as an underlying protocol. Therefore, the existing socket APIs will be used as supporting libraries for the implementation of the new API. The development of the software-based application will be performed in JAVA technology programming.

2. Area of Research

Although there are a variety of the applications used in today's networking scope, the underlying protocols of such applications are still either TCP or UDP. The API will be developed to support the problem mentioned above along with the development of the client and the server application for a particular scenario. This thesis will develop the API to be suitable for various applications using TCP or UDP as an underlying protocol. Ideally, it has to be developed to be suitable for all applications (e.g. FTP, Telnet or UDP-related applications), but for this thesis not all applications will be covered because of the different characteristics of each application. Each user-defined protocol has different states and parameters to be reconnected so that the thesis project will be focused on the service continuity based on the popularity of applications. The protocols we are researching are File Transfer Protocol (FTP), which uses TCP as an underlying protocol, and Real Time Protocol (RTP), which uses UDP as an underlying protocol. These two applications can be instances of the development for transport protocol. Certain features of real time applications, such as

synchronization, are not currently supported but are addressed as areas for further study.

IV. DESIGN AND IMPLEMENTATION

This chapter presents the critical parts of the project, including the APIs developed for persistent session support and the client-server application that utilizes the APIs. The details of the API and the GUI implementation are discussed. The application-user interactions are presented at the end of this chapter.

A. DESIGN

1. Main System Components

For the rest of this thesis, we will refer the APIs developed in this research to support persistent sessions *the Persistency API*. The application developed for this thesis has three major components: the server, the client and the persistency APIs. The details of each component are discussed below:

- Server – It is the data source in the communication and can handle multiple connections as needed. It needs to understand two types of requests, which is indicated in the initial connection message: the connection request and the reconnection request. The reconnection request contains more information than the first type. Based on the type of request and the information associated with it, the server determines the starting point of the data to be sent.
- Client – It provides the user interface and interacts with the user in order to meet the user requirement. It starts the reconnection process in the face of physical connection loss by calling the persistency APIs without user intervention. The user needs to

input the arguments required for the persistency APIs at the initialization.

- Persistency APIs – It supports the service continuity between the server and the client. It is the additional API derived from the existing library that the client needs to maintain persistent sessions for both TCP and UDP connections. It is defined as a class and can be called by the client when it detects a physical connection failure.

Persistency API is a major component in the application development and is used at the client side. The application runs in two modes, TCP and UDP. In the TCP mode, it transfers a file from server to client and can recover from temporary connection loss. In UDP mode, it transfers a video from server to client and can recover from temporary connection loss. In video file transferring, there are two types of protocols involved for the session. Real Time Streaming Protocol (RTSP) and Real Time Protocol (RTP). RTSP is used for video control session that uses TCP as an underlying protocol. RTP is used for retrieving the video packets which is one of UDP communications. The development is mainly focusing on the RTP session in order to provide the continuity of the video.

a. Graphic User Interface

Both client and server modules will provide a user-friendly interface. In this initial version the user interface of the client side will provide three main services:

- Protocol selection for the data transfer communication. The user must be able to select one of the transport protocols for the data transfer.
- View of the data transfer progress. A progress bar with the percentage of the file transfer session completed is implemented.
- View of the content of the file for both cases of text file and video file at the client side. These services are represented as separate panels to the user interface frame. Figure 8 shows a primitive version of a user interface at the client side.

For the server side, it is unnecessary to provide the user interface. Figure 9 shows an initial version of a graphical user interface (GUI) at the server side. The important GUI will be provided for the following purposes:

- View of the server operation for the connection process.
- View of the server operation during file transfer. This application will be provided in a separate panel.

Figure 8. Preview of the client's user interface

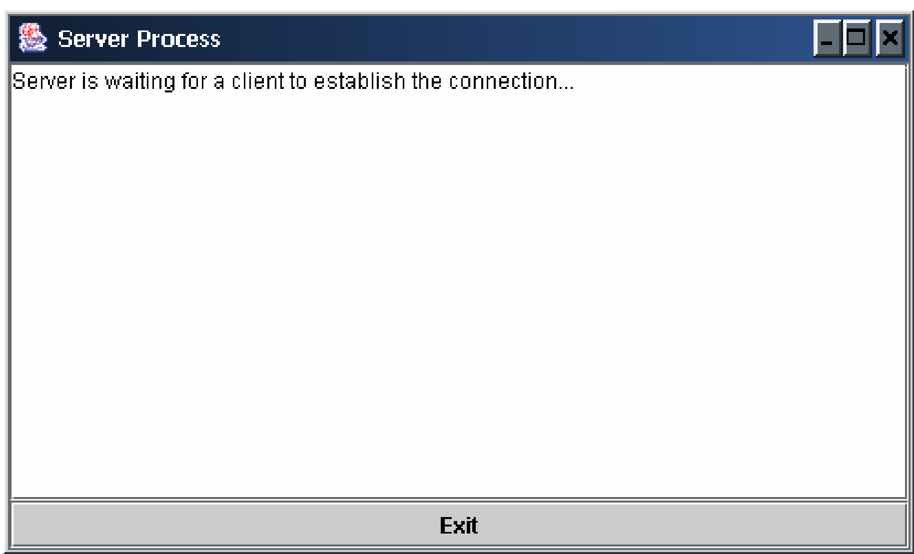

Figure 9. Preview of the server's user interface

2. Software and Development Tool

The test application and persistency APIs are written using the Java programming language. JBuilder9 Java editor environment is used to develop the application. For both

the server-side and the laptop/desktop client version, the Java 2 Standard Edition (J2SE) with the Java Development Kit 1.4 (JDK 1.4) is used. Persistency APIs for this application are developed based on existing Java libraries.

The following systems are needed to run the application with persistency API support:

- Server: Windows 98, NT, 2000 or XP operating system and a Java Virtual Machine.
- Client (desktop/laptop): Windows 98, NT, 2000 or XP operating system, Java Virtual Machine and the Java 2 SDK1.4 installed.

3. Basic API and Program Interactions

The user must specify the preferred type of transport protocol, which is either TCP or UDP, before starting the data transfer. For this thesis project, the selected transport protocol determines the type of application, file or video transfer, the client and the server will perform. As indicated in section A.1, selecting TCP will result in starting a file transfer application, and selecting UDP will result in starting a video transfer program. The specification of the type of the protocol must be done at the beginning of the execution.

a. File Transfer

The file transfer from the server to the client results from a client-side user request. The file size of a desired file is first transferred to the client so it can be used to display the progress bar. The server sends the file as a series of packets (arrays of bytes or arrays of frame) that the client collects until it receives all the content of the file. The client application keeps track of the number of packets received. In case of the physical

connection failure during the transfer, the persistency API, which resides at the client side, is used to reconnect until the physical connection is resumed. The new connection state is returned to the client, and the client continues to retrieve the rest of the packets until the end of the file. Thus, the client doesn't have to restart the connection and receive the content of the application from the beginning. If the physical connection is lost again, the same process will be repeated as before.

b. Client-server Communication Protocol

For the file transfer process, an application layer communication protocol must be established. The following figure shows the abstract of the communication protocol with persistency API support:

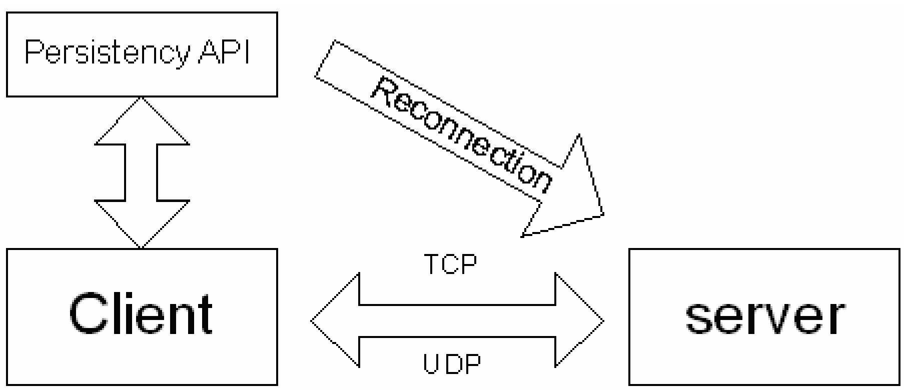

Figure 10. The communication scheme with persistency API support

The application starts with the server running and waiting for the client at either port 5555 for TCP or port 9999 for UDP connection. After the user starts the client application and makes a selection of the type of protocol to be used, the server will respond to the client connection request without any user intervention.

After an initial connection is established, a file size request will be sent followed by the file content request from the client to the server. The request packet field is shown in Figure 11:

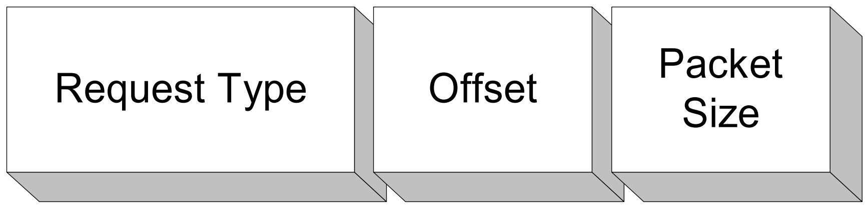

Figure 11. A request message packet

After receiving the first request from the client, the server sends the file size in bytes back to the client. Then it waits for the client’s next action. After getting the next request, the content of the file will be sent as a series of packets to the client.

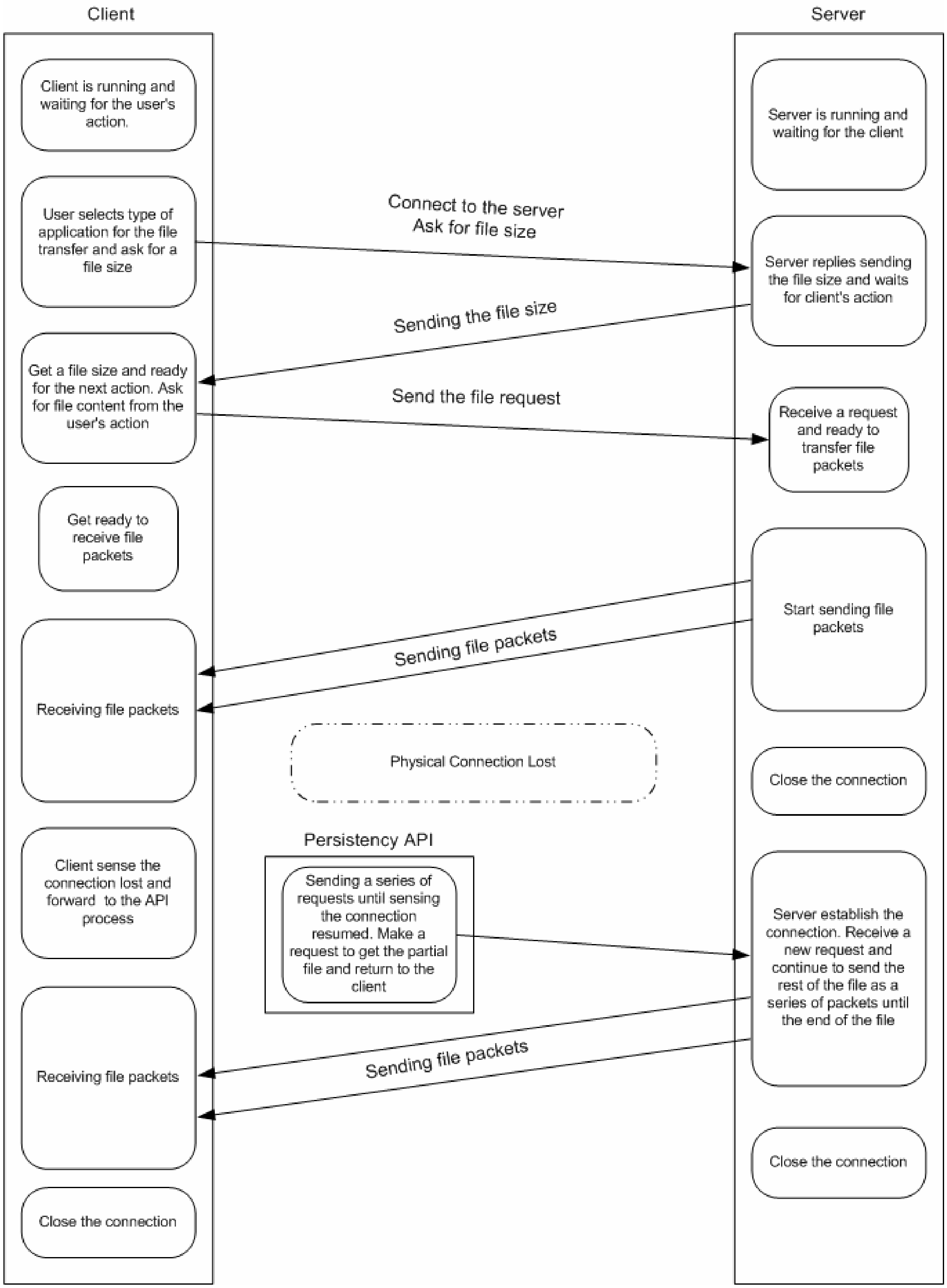

Figure 12. The communication protocol with API support

If, during the transmission, a physical connection failure occurs, the client side must detect it by using a time-out mechanism. A distinction between the packet loss and the physical connection failure can be judged by Java application exceptions. After detecting the failure, the client will call the persistency API to reconnect to the same server. A series of reconnect requests will be sent out until a reply from the server is heard. After the connection is resumed, information needed to reestablish a connection is sent to the server and the client's important parameters are being updated in order to continue the session. The client will continue to receive the rest of the data after the persistency API call is returned. Each physical connection loss causes the same partial transfer retrieval scheme to take place until the client receives all the data packets and both the client and server sides close the connection.

c. Activity Diagram

Figure 13 shows the activity diagram for API support application

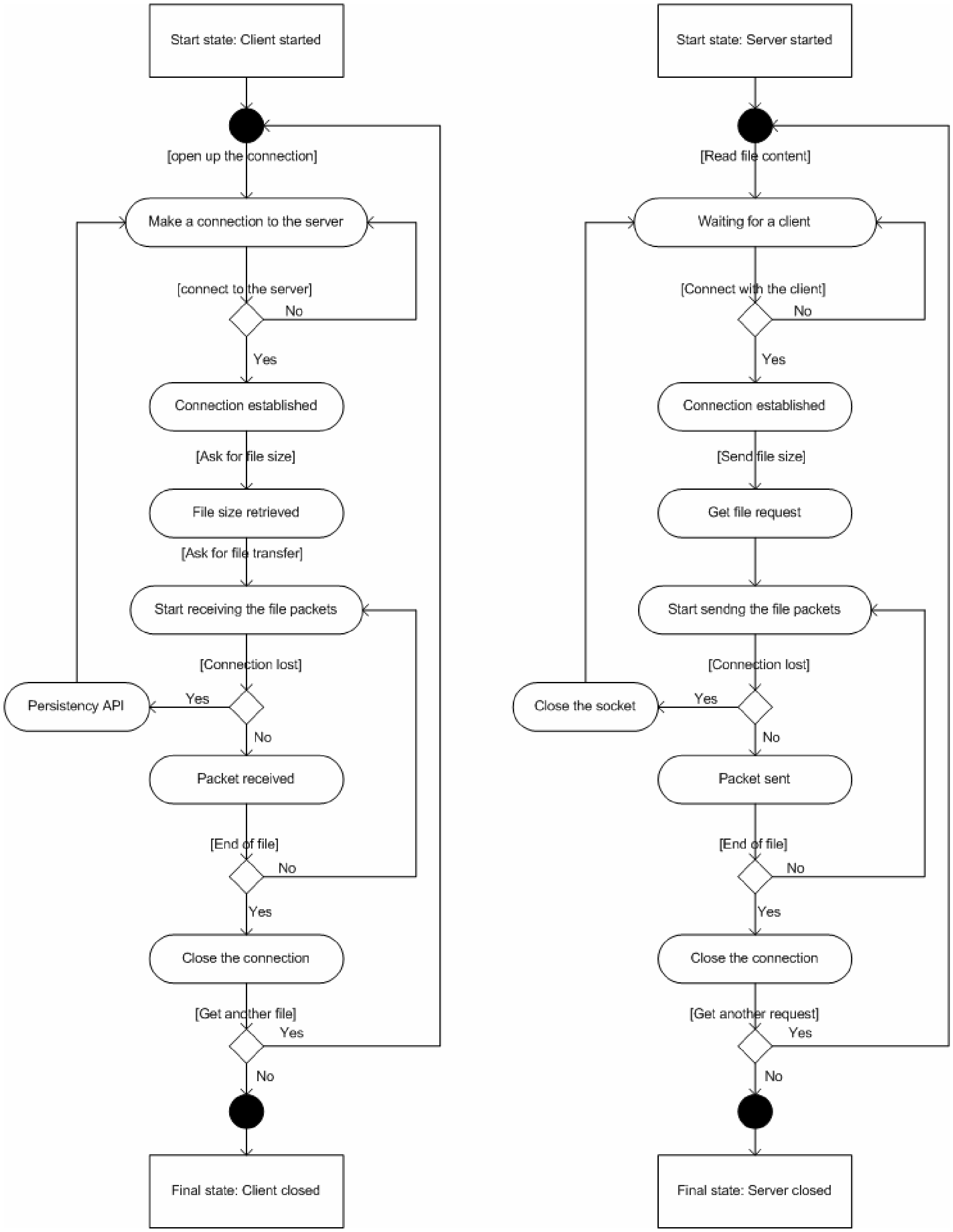

Figure 13. The activity diagram for persistency API

d. Class Diagrams

The following, Figure 14, shows the relationship between the classes for this thesis project:

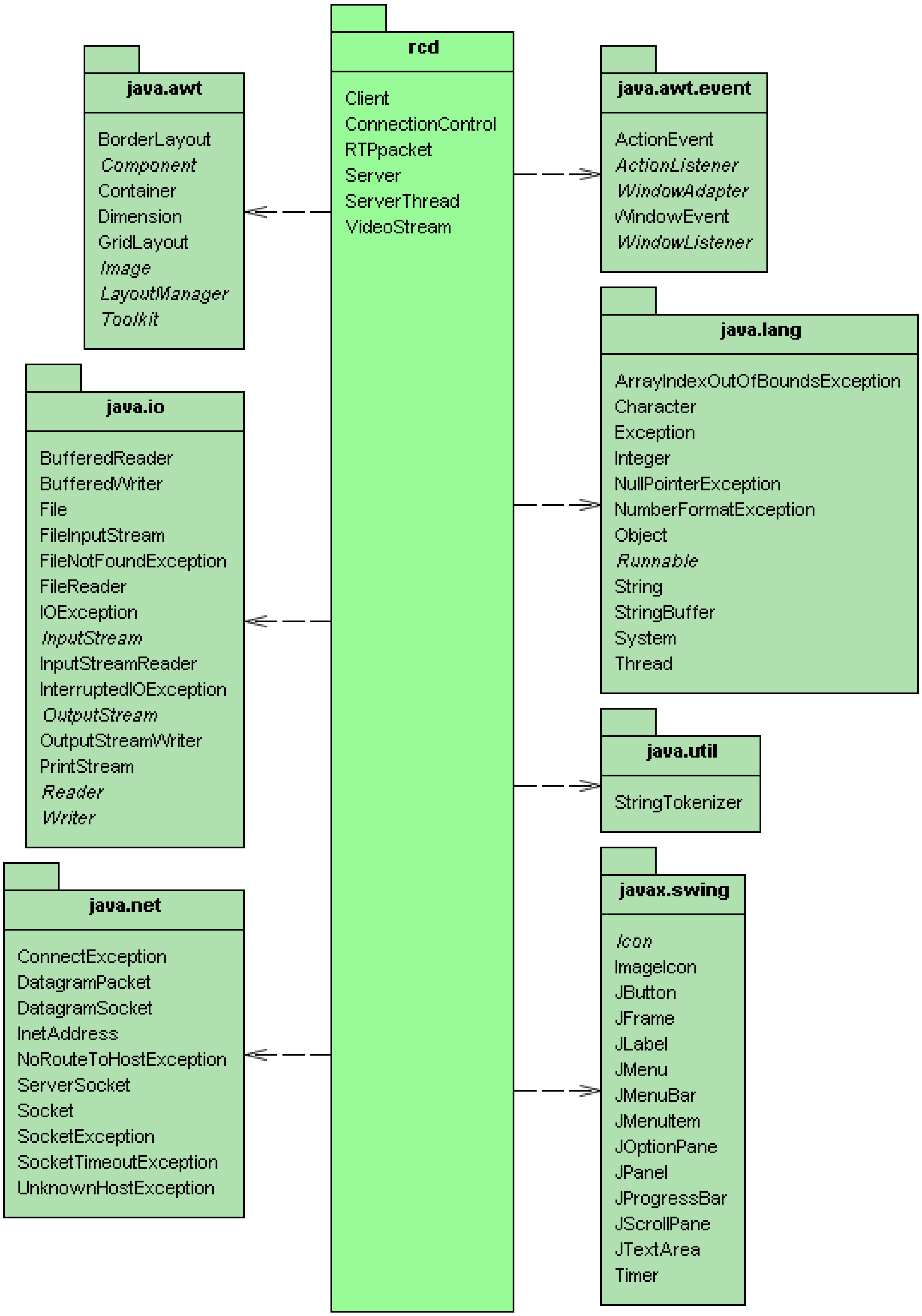

Figure 14. The class diagram for API support application

In addition, Appendix B shows the client desktop/laptop, the server and the API versions of the

class diagrams for the content of each class. Appendix B also includes the supporting classes for UDP sessions, which have RTPpacket and VideoStream classes at the client side.

B. PERSISTENCY API IMPLEMENTATION

The main purpose of the application developed for this thesis is to experiment and validate the persistent session support used. The specific API for such support, called the *persistency API*, is used at the client side. An instance of this class is one of client object members.

1. Overview

The 'ConnectionControl' class is created to provide persistent session support. It is coded to be an agent for executing the reconnection process on behalf of the client. This class has to be initialized at the first stage of the application execution, after the client object is created. It is also created as an object in order to cooperate with the client object. Therefore, the client has a particular function called *fileControl(Client client, int port, String host)* that initializes the *ConnectionControl* object. In the *fileControl()*, three parameters are used to initialize the persistency API object; *client* is the client object, *port* the initial communication port between client and server, and *host* the IP address of the server. The command for initializing the persistency API object is:

controlAgent = new ConnectionControl(Client client, int port, String host);

The controlAgent is an instance of the persistency API object. It needs to be initialized in order to coordinate with the client's object. For this initialization, there are three arguments at the first time of the initialization. These two latter arguments have to be set

in the initial values and may be updated later on. The ConnectionControl class has a critical function for the reconnection session called *ReconnectProcess()*. This function tries to connect to the server until it reestablishes the old session.

The important variables used in the reconnection session are the following:

- index – The parameter acts as offset of the data file sent so far. It is always updated implicitly during the data transfer session. It is retrieved automatically by the persistency API in order to initialize the reestablished session.
- host – The server IP address.
- port – The communication port for the file transfer. It is static for the TCP session but it is dynamic for the UDP session. The Java random function is used to generate the dynamic port number.
- client – The client object. ConnectionControl needs this handle to pass control back to the client after connection is reestablished. The client will continue the data transfer process.
- done – It is the logic to control the iteration in order to make a new connection to the server. It will be set to another value after the physical connection is resumed.

The main function of this persistency API, function *reconnectProcess()*, is being called to do the following:

- Obtain the type of underlying protocol and the offset of the file from the client at the point before the connection failure.

- Send the new requests to the server until the physical connection is recovered.

The additional functions in this persistency API class are supposed to support the reconnection operation. There are composed of the following:

- initialization() – This function initializes the critical data members of the class
- setIndex(int index) – This function sets the offset value of the desired file. It is being called during each packet transmission.
- open() – This function is the first step after the resumed connection. It creates the input and output streams in order to communicate with the server.
- sendTCPRequest() and sendUDPRequest() – These two functions are the final steps of the persistency API. The details are discussed in the next section.

2. The Use of Persistency API

When the client detects the physical connection failure via the Exception in Java, it calls the member function *reconnectProcess()* of ConnectionControl:

```
controlAgent.reconnectProcess(String);
```

The argument for this function is a string, used to indicate the underlying protocol. It can have the value "TCP" or "UDP", depending on the type of transfer application that is being used. This argument is important to determine the appropriate new request to the server. The following is the code segment for the reconnectProcess() function:

```
114  public void reconnectProcess(String type) {
       ...
138    while (!done) {
140        try {
145           socket = new Socket(host, port);
              ...
152           open();
155           client.setParameters(socket, br, bw);
157        if (type.equalsIgnoreCase("TCP")) {
             ...
160          sendTCPRequest();
162        } else {
164          try {
167            sendUDPRequest();
169          } catch (Exception e) {}
171        } // end if - else
173        done = true; // set the exit of the loop
175      } catch (UnknownHostException uhe) {
           ...
179      } catch (IOException ioe) {
           ...
187      } // end try - catch
191    } //end while
193  } // end reconnectProcess()
```

After being called, the persistency API starts a job by sending a series of requests to the server. If there is no reply, the new socket, line 145, will not be created and yields the result in exception occurrence. In Java, if there is an exception occurrence, the appropriate catch statement will handle this error. Since it jumps to the catch statement, on line 179, due to the socket

establishment failure, it assumes no connection is resumed and continues the iteration within the while statement, line 138 - 191. After the connection is resumed, the persistency API continues by opening the new stream between the client and the server, in line 152, and then sets the important parameters to the client object in order for the client to continue the data transfer after the reconnection completes. Next, based on the argument passed in the function, a corresponding request, in line 160 or 167, is sent to the server. After sending a request to get a partial file from the server (the appropriate offset has to be sent), this persistency API returns to the client. The client will continue retrieving the partial date at the point it lost the connection.

For clarification, after establishing a new connection with the server (below line 145), the further process in which the API supports the recovery of the file transfer can be categorized into two groups as follows:

a. Using Persistency API for TCP

After the new connection is established, the persistency API is trying to make a request to get the partial file. Therefore, there are two types of applications. In this case, the text file is being retrieved. From the code segment shown earlier, if the main function of the persistency API, reconnectProcess(), receives “TCP” as argument, it will force the ‘if’ statement to call *sendTCPRequest()*, on line 160, to get the text file and the file transfer session will be continued right after this command. The following is the pseudo code used for sending the new TCP request to the server:

```
238  private void sendTCPRequest() {
```

```
240    String textOut = "/get " + index;
       ...
247    client.send(textOut);
249  } // end sendRequest()
```

The parameter 'index' is already updated so this function will use it automatically. It is an offset of the desired file where the server should start the transfer in the reestablished connection. The function *sendTCPRequest*() uses the client's existing function to send the request after establishing the new connection with the server in order to have the client continue the file-retrieve job after the control is returned to the client. For the TCP case, API returns to the client immediately after sending the new request. It assumes that the time of returning to the client is faster than the period of the request, which is sent to the client combined with the period packets that travel from the server to the client. The service continues from the point it returns and the client continues to receive the rest of the file.

b. Using Persistency API for UDP

For the case that reconnectProcess() function gets "UDP" as an argument from the client, the 'if' statement of line 157 will evaluate the logic to false and *sendUDPRequest()* on line 167 will be called. *sendUDPRequest()* is the function needed to send the new request for UDP session. The following is the code segment used for sending the new UDP request to the server:

```
256  private void sendUDPRequest() throws Exception {
       ...
262    int newRTPPort = client.randomRTPPort();
       ...
```

```
267    RTPsocket = new DatagramSocket(newRTPPort);
269    client.setRTPSocket(RTPsocket);
270    client.setRandomRTPPort(newRTPPort);
       ...
273    client.send("/Setup " + index + " " +
       newRTPPort);
274    client.send_RTSP_request("PLAY");
       ...
278    client.timer.start(); // continue the timer
                         after sending the new request
       ...
282  } // end sendUDPRequest()
```

Line 262 generates a new RTP port for receiving the partial video file from the server. In order to avoid the packets from different sessions colliding at the same port, a random function is used to generate a new port number. The new RTP socket using the new RTP port needs to be assigned to the client object in order to achieve service continuity because the control will return to the client object after a new connection to the server is established.

The parameter ‘index’ is also an offset of the desired file and already updated. Java timer class, which is used control the file transfer of the video packets, is stopped when the connection failure occurred. Thus, this function needs to be reactivated after the connection is resumed for the next video packets transfer session. As a result, it has to be called again after the connection is resumed. After the new request from the API has been sent, the API returns the functionality back to the client by using the *start()* method, line 278 of *ReconnectProcess*(),

in which the Java timer is reactivated. The service continues from the point it returns and the client continues to receive the rest of the file.

C. APPLICATION USAGE GUIDE

1. Client

The client starts the application by waiting for actions from the user. The user must select one of the options from the drop down menu as shown in Figure 15.

Figure 15. Client's selection panel

After the button *connect* is pressed after choosing the selection, the client program automatically connects to the server. The client first sends the file size request. After getting the file size, another panel will show up and wait for the user's next actions to retrieve the file.

After the user presses the "Get Data" button for TCP connection or the "Play" button for UDP connection, the

client starts to receive the file packets from the server and displays the content on the panel until the end of the file. Figures 16 and 17 show the preview of the second panel for TCP and UDP respectively.

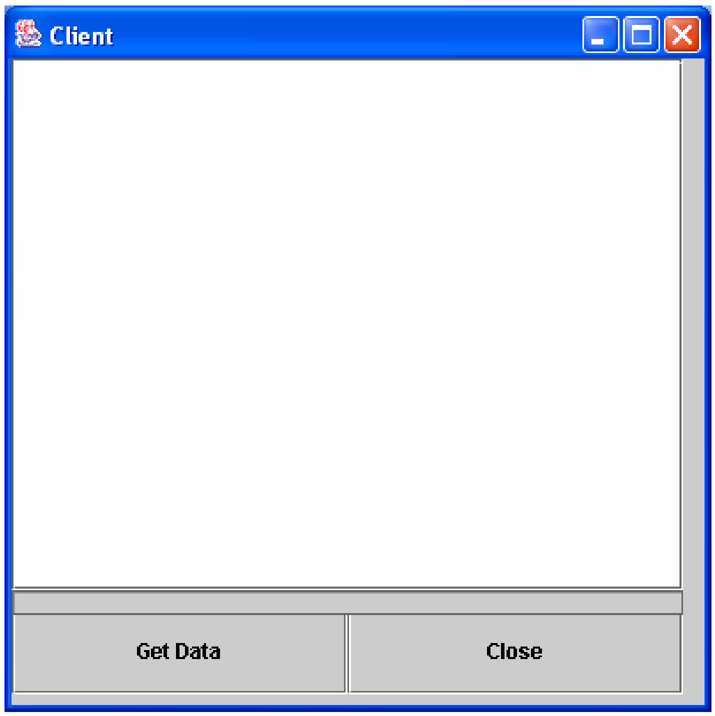

Figure 16. Client's second panel for TCP session

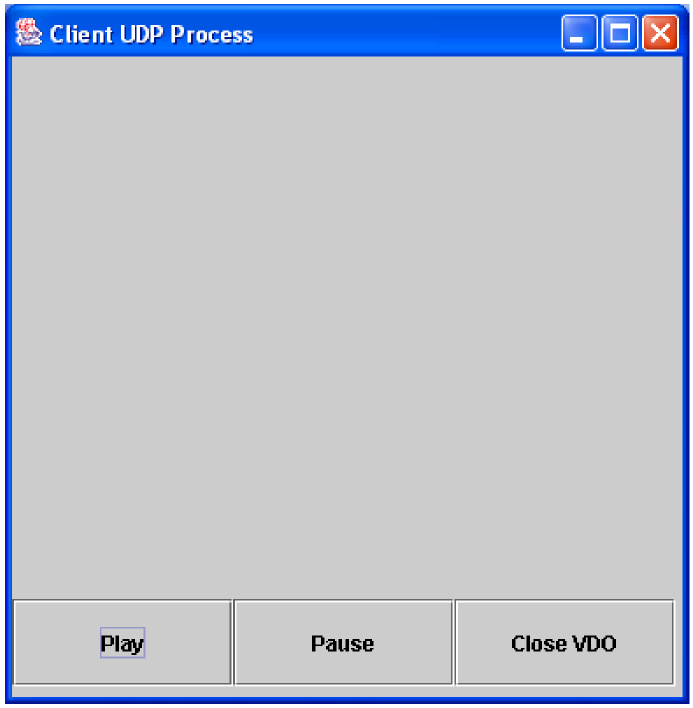

Figure 17. Client's second panel for UDP session

The following figures show the second panels while retrieving the data from the server.

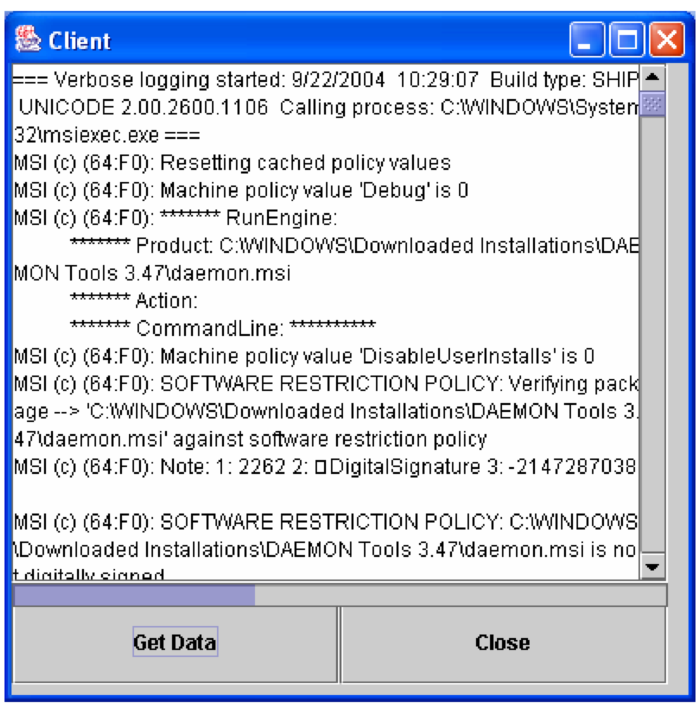

Figure 18. Client's second panel during TCP transmission

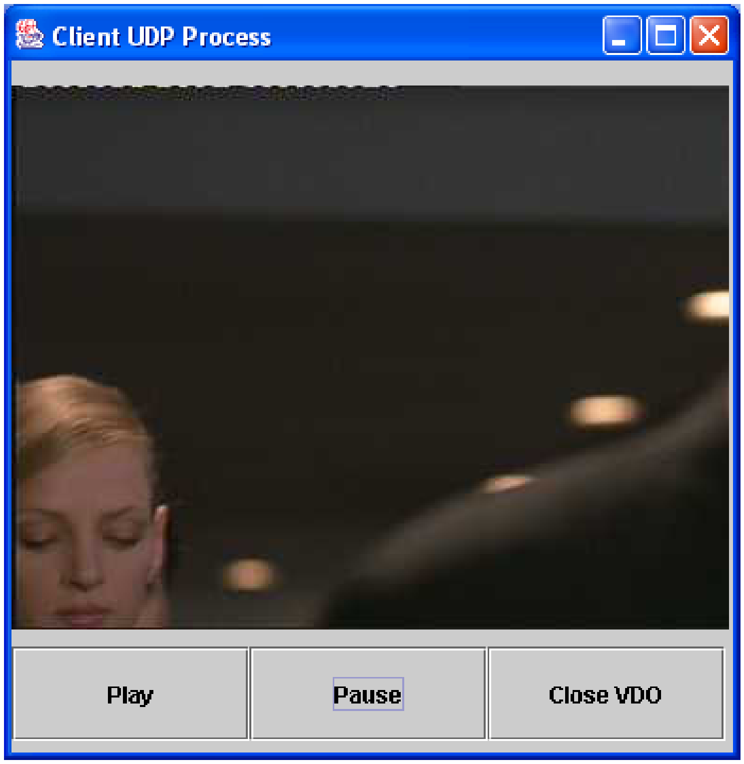

Figure 19. Client's second panel during UDP transmission

If any connection failure occurs, the second panels will be paused automatically and the reconnection process is undertaken without any user interaction. When the connection is reestablished, this panel resumes showing the content of the file from the point it paused.

2. Server

The server starts the application by waiting for the client to connect to it. After establishing the connection, a server thread is created to handle the transfer session with the client. The main server process, in the mean time, can accept other client connections (maximum number of clients is 30). The following figures indicate the server process:

Figure 20. Information message from the server

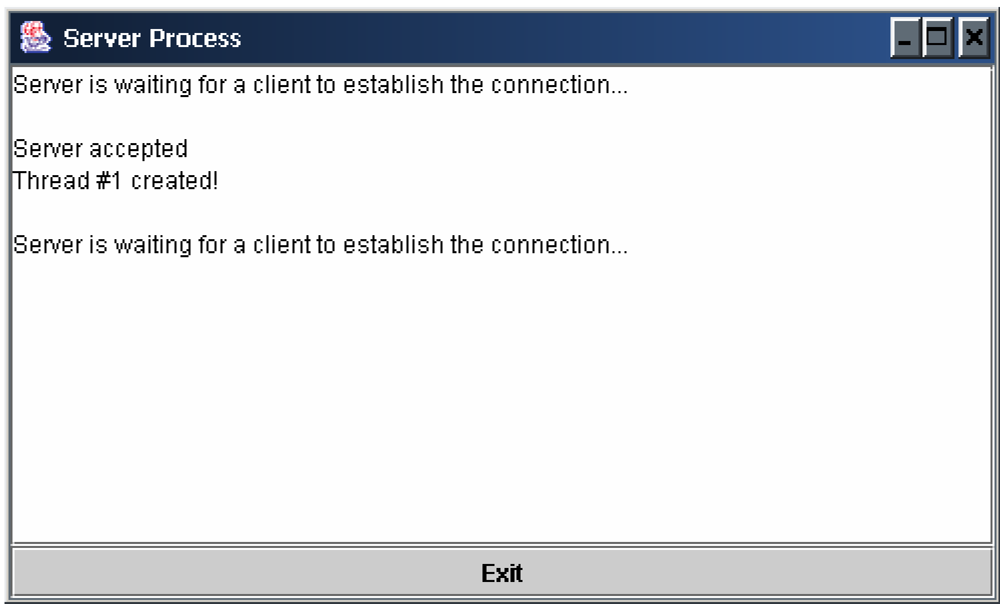

Figure 21. The server process

After being connected, another information panel is popped up in order to show the process of the transmission. These panels are displayed and updated throughout the process until the end of the file. The following figures show the transmission process using TCP and UDP as an underlying protocol respectively:

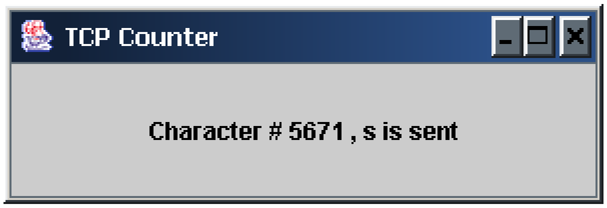

Figure 22. The server's transmission process for TCP session

Figure 23. The server's transmission process for UDP session

V. TESTING

This chapter describes the testing of the data transfer application developed, including the test network description, the various scenarios that were used, and the general results of the testing.

A. TESTING NETWORK DESCRIPTION

The testing of the API support application required the installation of a basic network that simulates several scenarios in which all the application components' operations can be tested.

1. Practical Considerations and Limitations

- Home-based wired networking was used for data transmission between end users to simulate a small network scenario.
- Both wired and wireless network at the Naval Postgraduate School (NPS) were used to simulate the Internet network environment.
- During the test, the IP address of the server was assumed to be static.
- A sufficient number of wired clients and wireless enabled devices were used to test the requirements of the thesis research. It is not our goal to test the volume of client traffic that the application can handle.
- In most of the testing scenarios, the additional "delay" during transmission is added to reduce the speed of the file transfer and the reconnection process. That means the server's and client's program response was slowed by the use of a "delay"

function so that it is easier for the user to observe the communication protocol features and behavior during the test scenarios.

- The connectivity failures necessary to test the protocol responses were manually caused by either unplugging the network connection in the wired devices (mainly servers), or by disabling or removing the wireless adapters from the wireless enabled client devices.

2. Testing Network

As Figure 24 shows, for the small network scenario, the testing network consists of a server, a client and a switch connecting directly for a wired environment. For a wireless environment, they are connected via wireless adaptors through a router that also connects to the Internet. The server in both the wired and wireless setup has the static IP address.

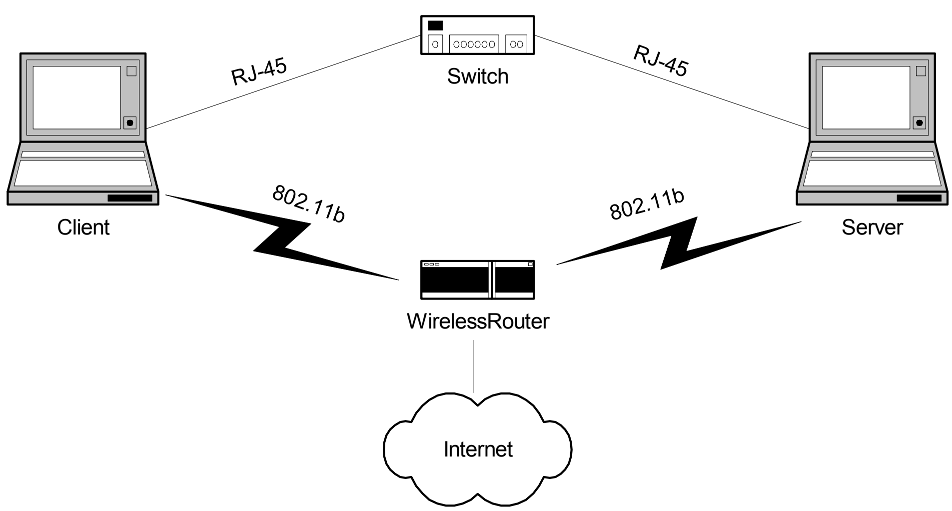

Figure 24. The home-based networking architecture

For the simulated WAN network, the client is connecting to the server using wired connection via a switch, and wireless connection via a wireless router. As shown in Figure 25, the client is connected to the server via the NPS network for both wired and wireless WAN testing.

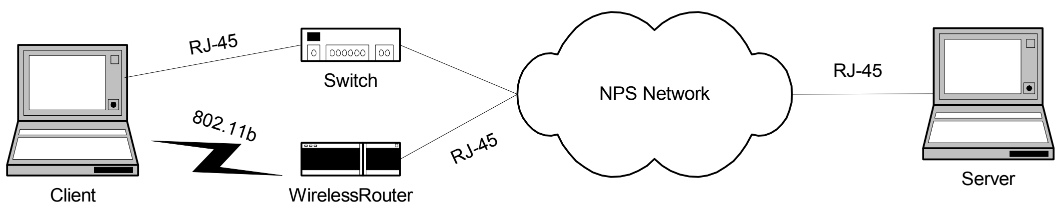

Figure 25. The NPS-based WAN network setup

B. TESTING SCENARIOS

Several tests were developed to emulate the various problems that might be encountered while running the application. These scenarios are used to ensure that the goals of the thesis research were fulfilled. The following descriptions list all the testing scenarios and associate them with a reference code so that they can be referred later in the result description without naming them explicitly. The reference codes start with a group of letters that indicates the general scenario type and ends with a counter number. The reference code part that refers to the scenario's type is one of the following:

- SUIS : Server User Interface Scenario. Situations that can happen during the interaction of the server's user with the available user interface.
- CUIS : Client User Interface Scenario. Situations that can happen during the interaction of the client's user with the available user interface.

* APIS : API Scenario. Situations that API does during the establishment of the new connection for the client to the server.

1. Scenario Reference Code and Scenario's Description

SUIS-1	The server accepts the connection and waits for the request from the client.
SUIS-2	The server sends the data packets in a TCP session.
SUIS-3	The server sends the video packets in a UDP session.
SUIS-4	The server accepts the new connection and disregards the previous connection.
CUIS-1	Failure in the type of protocol selection when it is necessary to select the protocol used for transfer.
CUIS-2	Starting download in a TCP session.
CUIS-3	Starting download in a UDP session.
CUIS-4	Detecting a physical connection lost while downloading the packets from the server.
APIS-1	During the file transfer, the connection is lost but is restored again after a short period of time. The *ConnectionControl* object performs the task instead of the client.
CUIS-5	Display confirmation to the user before closing the current session.

C. TESTING RESULTS

This section explains how the network components responded to each of the scenarios and how the user interface helped the user to be informed if a file transfer

failed during its operation. The reference codes listed in Section A are used to refer to each scenario.

- **SUIS-1.** The server accepts the connection and waits for the request from the client. The result was that the program informed the user via a message as follows:

Figure 26. The server establishes the connection

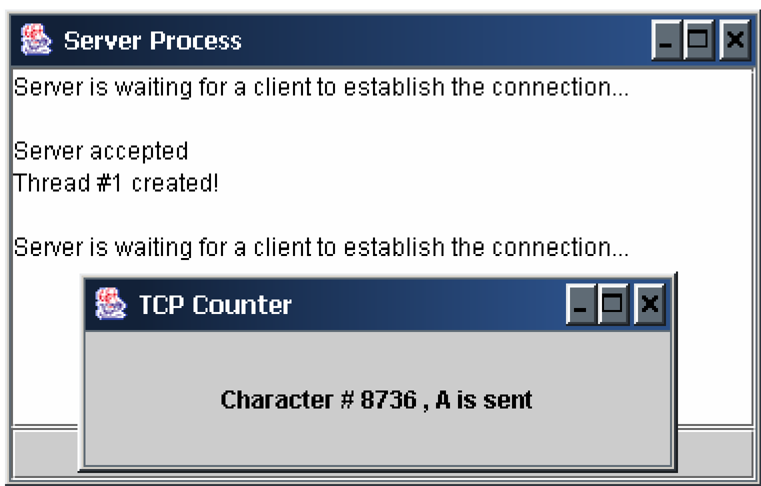

Figure 27. The server is ready for the TCP file transfer

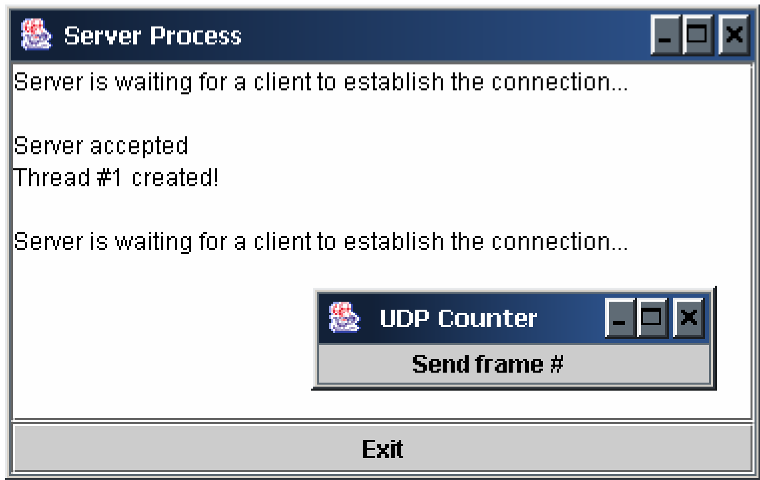

Figure 28. The server is ready for the UDP file transfer

- **SUIS-2.** A panel displaying the text sent informs the user of the progress for a TCP session.

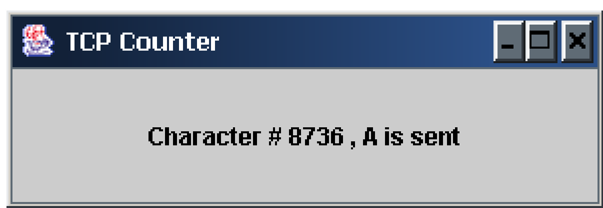

Figure 29. The proxy server process for a TCP session

- **SUIS-3.** A panel displaying the frame number sent informs the user of the progress for a UDP session.

Figure 30. The proxy server process for UDP session

- **SUIS-4.** This is the result from a lost connection. The server accepts the new connection and disregards the previous connection. The new thread is created to handle the new communication.

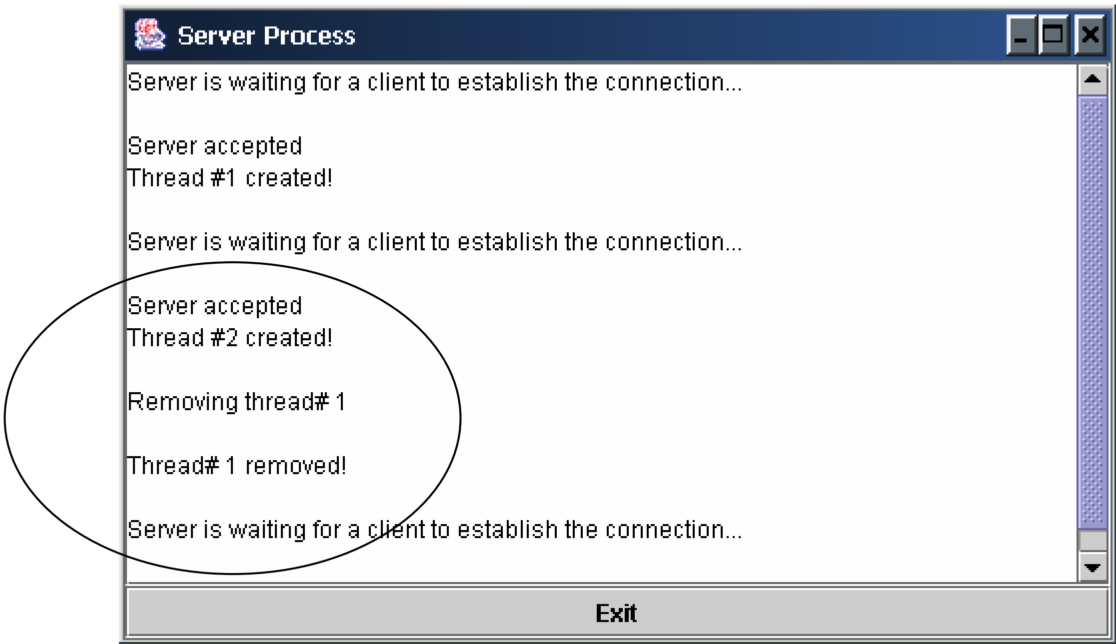

Figure 31. The server process after new connection

- **CUIS-1** The client user needs to select the type of protocol to be transferred in order to avoid an error. The panel in Figure 32 appears if the user tries to connect before selecting the type of protocol.

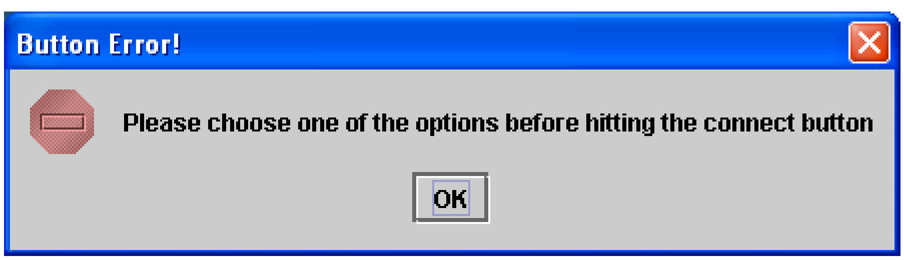

Figure 32. The response of client side for CUIS-1

- **CUIS-2.** A window displays the content that the client received during a TCP session (as shown in Figure 18 in chapter 4 section B.2.a).
- **CUIS-3.** A window displays the video content that the client received during a UDP session (as shown in Figure 19 in chapter 4 section B.2.a).

- **CUIS-4.** When connection failure is detected, a GUI is displayed to alert the user, and user action is needed for the client to begin the reconnection process. The following figures show the sequence of panels displayed to the users at the client side:

Figure 33. The information message of the client (1)

Figure 34. The information message of the client (2)

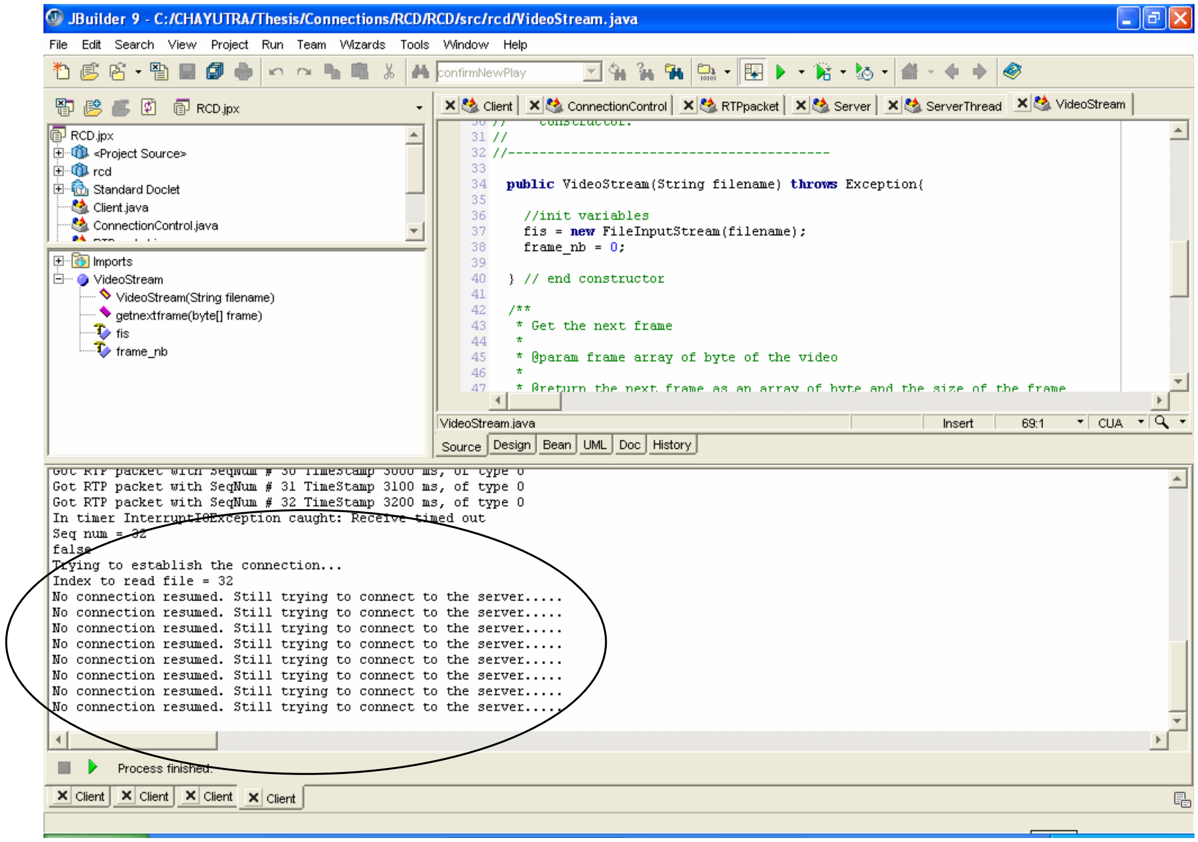

Figure 35. The persistency API's reconnection process

- **APIS-1** During the file transfer, the connection is lost but is restored again after a short period of time. The client attempts to reconnect to the server by calling the persistency API. The image progress panel of the user interface, which displays the image or file download progress, stops updating until the client reconnects to the server and begins retrieval of the rest of the file. Figure 36 shows the attempt of the API program to connect to the server as well as the paused state of the user interface during connection failure. Figures 37 and 38 show the message when the connection is resumed, followed by the messages showing that the process is continued from the point where the connection was lost.

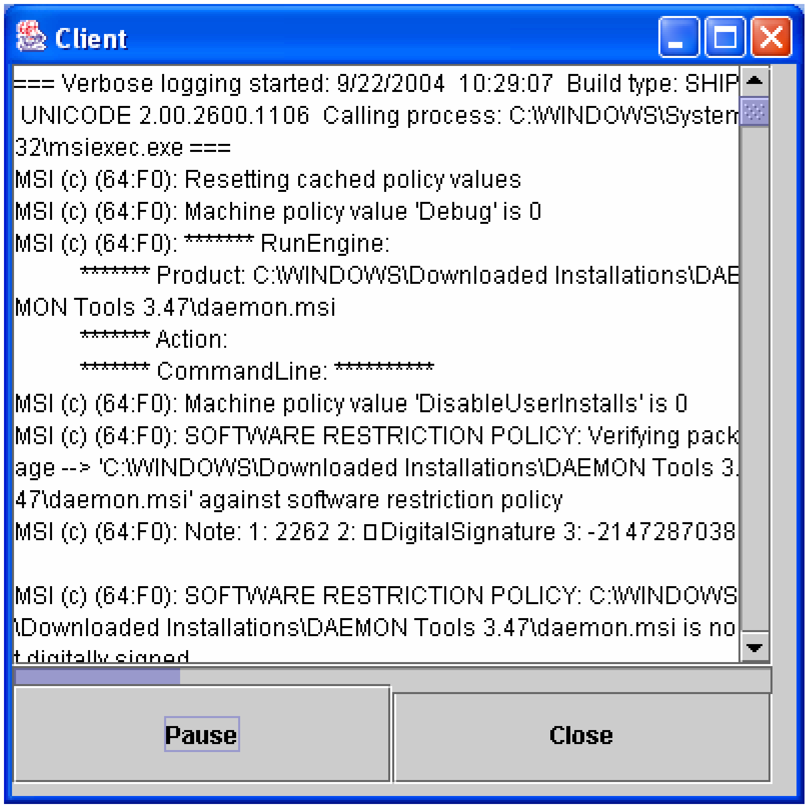

Figure 36. The client's user interface during connection failure

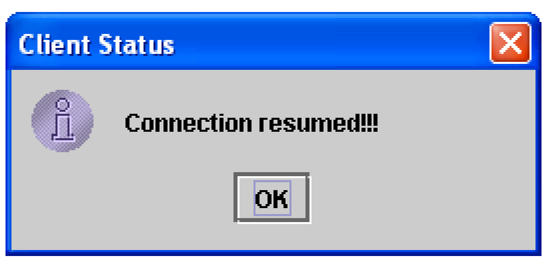

Figure 37. The message from API showing the status (1)

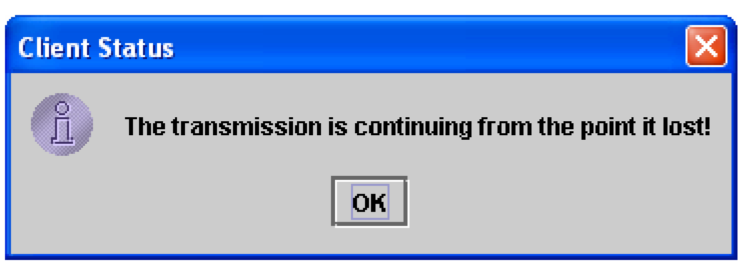

Figure 38. The message from API showing the status (2)

- **CUIS-5** There are two panels at the client side running at the same time: the client's control panel (Figure 16) and the client process panel (Figure 18 for TCP or Figure 19 for UDP). Both panels are waiting for the action from the user. While the client process is idle or still running, if the user wants to quit at any time, the application will ask for confirmation before leaving the application. The action from pressing the "close" button from one of the client's panels will yield the result shown in Figure 39.

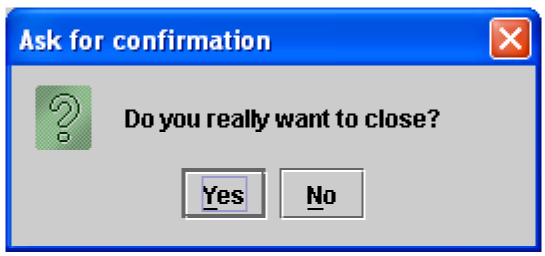

Figure 39. The final confirmation message

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION AND FUTURE WORKS

A. SUMMARY

The goal of this thesis research was to design an API to support persistent session services to various applications using TCP or UDP as an underlying protocol, and to implement an application that uses and demonstrates the operation the persistent session service. Other proposed file transfer protocols for service continuity were examined to study their characteristics and features used to recover from a connection failure.

The communication program implemented is a client-server application that supports multithreading on the server side and can dynamically recover from data transfer failure due to intermittent physical connection loss. This file recovery feature is achieved by designing and implementing the API at the client side. The *ConnectionControl* object, when called by the client application, tries repeatedly to connect to the server and resume the data transfer session from the point where connection failure occurs.

The application was designed with user interfaces that make the dynamic partial file retrieval visible to the user in real time. Special user interface panels are created to show the file transfer progress and display the data received.

The main scenario tested during the communication application testing was when the connection failed during the file transfer and the client program successfully reconnected when the physical connection was restored. Our test also validated the file management and the

multithreaded behavior at the server side. Both wired and wireless network environments were used in testing at the client side. Users at the client side were able to visualize the file transfer progress and control the file transfer options (request file, stop downloading, or choose to continue previous failed file transfers).

B. FUTURE WORK

Extending the research scope of this thesis and the application developed in support of it, there are issues that raise opportunities for further research. They include:

1. Communication Protocol Design

Based on the APIs developed in this thesis for specific applications, further research could be focused on enhancing the API to have more capability. Some key areas towards this direction could be:

- Support a fully mobile networking environment. Further enhancement in the API can be supported for portable devices with respect to data session survivability in a wireless environment. The API may have more capability but should be small enough to be more suitable for mobile device.
- Support for migration of UDP sessions. As discussed in Chapter II, migration in TCP has been studied. Further research can be done on enhancing the service continuity for both kinds of protocol.
- Lower level API. The API may be written using the concept developed in this thesis but implemented at the lower level of the network stack to achieve better performance.

2. Application Development

In application-level development, the following further work can be done:

- Extending the persistency API class library. Create a class or additional API library for the persistent connection protocol that application developers can use to develop new applications. This proposed API should be able to support generic applications at the application layer, e.g., it supports all applications using either TCP or UDP as the underlying protocol.
- Developing additional applications that use persistent data sessions. Examine other types of applications where applying the persistent data sessions can be useful.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. CLASS SOURCE CODE

```
/**
 * Title: API Development for Persistent Data Sessions
   Support
 * Description: Application client
 * Compiler : JBuilder 9
 * Author CPT.Chayutra Pailom THA
 * Date : January 20, 2005
 */

import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.Timer;

/**
 * This is part of the connection process which is connect
   to the server.
 * This client is intended to connect and receive the data
   from the server
 * using two different protocols; TCP and UDP. This class
   is intended to
 * use socket programming for both types of protocols and
   graphic user
 * interface on order to interact with the user.
 *
 * Expected server protocol support: Both TCP and UDP
   applications
 *
 * @author CPT Chayutra Pailom THA
 */
public class Client extends JFrame implements Runnable {

    //-------------------------------------------------------
    //
    //                Data Members:
    //
    //-------------------------------------------------------

    //---- Global variables for TCP or UDP process ----/

    /** Runnable TCP or UDP object */
    private Thread TCPThread, UDPThread;
```

```
/** The physical connection control process */
private ConnectionControl controlAgent;

/** Socket for TCP and RTSP request */
private Socket socket;

/** The buffer for input stream */
private BufferedReader br;

/** The buffer for output stream */
private BufferedWriter bw;

/** The IP address of the server */
private String host;

/** Port to communicate with the server */
private int port;

/** Logic to control the redundant data manipulation */
private boolean knowSize;

/** Logic for doing TCP or UDP process */
private int connectionCase;

/** The critical variable for reconnect process for
     both TCP and UDP */
private int index;

/** Connection status */
private boolean connect;

//------------- GUI for TCP --------------/

/** JFrame for TCP process */
private JFrame k;

/** Container to be added onto main frame */
private Container container;

/** The status bar showing how far of TCP process */
private JProgressBar progressBar;

/** The area to show the process and the TCP data */
public JTextArea textArea, textArea2;
```

```
/** The button to shutdown the visibility of the GUI */
private JButton buttonClose, buttonClose3;

/** The button to start retriving the data from the
    server */
public JButton buttonConnect, buttonData;

/** The panels for containing small functions of GUI */
private JPanel TCPPanel, buttonPanel, buttonPanel3,
        textPanel, textPanel2;

/** The container for menu commands */
private JMenuBar menubar;

/** The container for menu items */
private JMenu menuCommand;

/** The function for scrolling the text area */
private JScrollPane scrollPane, scrollPane2;

/** The items to be chosen for data manipulation */
private JMenuItem size, filePersistent,filePersistent2;

//------------ GUI for UDP ---------------/

/** The main frame of the UDP process */
private JFrame f;

/** The button to shutdown the visibility of the GUI
    for UDP */
private JButton buttonClose2;

/** The button to setup and start the video process */
private JButton playButton;

/** The button to pause the video process */
private JButton pauseButton;

/** The panel for containing the small functions for
    UDP GUI */
private JPanel mainPanel;

/** The panel contained functional video commands */
private JPanel buttonPanel2;

/** The label for the video */
private JLabel iconLabel;
```

```
/** The video to be shown */
private ImageIcon icon;

//--------------- TCP Variables ------------------/

/** Size of the desired data used for progress bar */
private int fileSize;

/** The timeout constant for socket */
private final int SOCKET_TIMEOUT = 5000;

//-------------- RTSP variables -----------------/

/** Boolean stand for the state */
private boolean ready;

/** Sequence number of RTSP messages within session */
private static int RTSPSeqNb = 0;

/** End of command */
private final static String CRLF = "\r\n";

//---------------Video constants-----------------/

/** RTP payload type for MJPEG video */
private static int MJPEG_TYPE = 26;

//--------------- RTP variables -----------------/

/** UDP packet received from the server */
private DatagramPacket rcvdp;

/** socket to be used to send/receive UDP packets */
private DatagramSocket RTPsocket;

/** port where client will receive the RTP packets */
private static int RTP_RCV_PORT = 9999;

/** new port where the client will receive the RTP
    packets */
private int newRTPPort;

/** timer used to receive data from the UDP socket */
Timer timer;
```

```
/** buffer used to store data received from server */
byte[] buf;

//-----------------------------------------------------
//
//                  Constructor:
//
//-----------------------------------------------------

/**
 * Default constructor
 *
 * @param pHost - IP address of the server
 * @param no - port number of the server to be
              connected
 */
public Client(String pHost, int no) {

    // Variables initialization
    host = pHost;
    port = no;
    knowSize = false;
    connectionCase = 0;
    connect = false;

    /** Initialize the control process object.
     * must call in order to avoid incomplete physical
       connection */
    fileControl(this, port, host);

    //--------- Client GUI Process -----------/

    // Main frame attributes
    container = getContentPane();
    container.setLayout(new BorderLayout());
    this.setTitle("Client");
    this.setSize(400, 400);
    this.setLocation(500, 0);

    // buttons,labels and panels
    buttonClose = new JButton("Close");
    buttonConnect = new JButton("Connect");
    buttonPanel = new JPanel();
    textPanel = new JPanel();
    buttonPanel.add(buttonClose);
    buttonPanel.add(buttonConnect);
```

```
// menu items for the client commands
size = new JMenuItem("Get Size");
filePersistent = new JMenuItem("Get File TCP
   PERSISTENT");
filePersistent2 = new JMenuItem("Get File UDP
   PERSISTENT");

// menu for the items to be added
menubar = new JMenuBar();
menuCommand = new JMenu("Command");
menuCommand.add(size);
menuCommand.add(filePersistent);
menuCommand.add(filePersistent2);
menubar.add(menuCommand);
this.setJMenuBar(menubar);

// Initialize handlers
ButtonHandler buttonHandler = new ButtonHandler();
MenuHandler menuHandler = new MenuHandler();

// action listeners for buttons
buttonClose.addActionListener(buttonHandler);
buttonConnect.addActionListener(buttonHandler);

// action listeners for menu commands
size.addActionListener(menuHandler);
filePersistent.addActionListener(menuHandler);
filePersistent2.addActionListener(menuHandler);

// text components
textArea = new JTextArea();
textArea.setLineWrap(true);
textArea.setEditable(false);
scrollPane = new JScrollPane(textArea);;

// border layout placements
textPanel.setLayout(new BorderLayout());
textPanel.add(scrollPane, "Center");
container.add(textPanel, "Center");
container.add(buttonPanel, "South");

setVisible(true); // set the visibility of the main
                     GUI

textArea.append("Client is ready\n");
textArea.append("Please choose one of the commands
    at the command bar\n");
```

```
//------------- TCP GUI Process -----------/

// frame for TCP GUI
k = new JFrame("Client TCP Process");

// add frame window attribute
k.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

// TCP process frame attributes
k.setTitle("Client");
k.setSize(400, 400);
k.setLocation(0, 0);

// buttons and panels
TCPPanel = new JPanel();
buttonPanel3 = new JPanel();
textPanel2 = new JPanel();
buttonData = new JButton("Get Data");
buttonClose3 = new JButton("Close");

// action listeners for buttons
buttonClose3.addActionListener(buttonHandler);
buttonData.addActionListener(buttonHandler);

// text components
textArea2 = new JTextArea();
textArea2.setLineWrap(true);
textArea2.setEditable(false);
scrollPane2 = new JScrollPane(textArea2);
progressBar = new JProgressBar();

// Add buttons into button panel
buttonPanel3.setLayout(new GridLayout(1, 0));
buttonPanel3.add(buttonData);
buttonPanel3.add(buttonClose3);

// border layout placements
textPanel2.setLayout(new BorderLayout());
textPanel2.add(scrollPane2, "Center");
textPanel2.add(progressBar, "South");
TCPPanel.setLayout(null);
TCPPanel.add(textPanel2, "Center");
```

```
TCPPanel.add(buttonPanel3, "South");

// panel attributes
textPanel2.setBounds(0, 0, 380, 315);
buttonPanel3.setBounds(0, 310, 380, 50);

// Add TCP main panel into JFrame
k.getContentPane().add(TCPPanel,
   BorderLayout.CENTER);

//------------- UDP GUI Process ---------------/

// frame for UDP GUI
f = new JFrame("Client UDP Process");

// add frame window attribute
f.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

// buttons and panel initialization
playButton = new JButton("Play");
pauseButton = new JButton("Pause");
buttonClose2 = new JButton("Close VDO");
mainPanel = new JPanel();
buttonPanel2 = new JPanel();

// Add buttons into button panel
buttonPanel2.setLayout(new GridLayout(1, 0));
buttonPanel2.add(playButton);
buttonPanel2.add(pauseButton);
buttonPanel2.add(buttonClose2);

// Add action listener for each button
playButton.addActionListener(new
   playButtonListener()); // create object play
pauseButton.addActionListener(new
   pauseButtonListener());// create object teardown
buttonClose2.addActionListener(buttonHandler);

// Image display label
iconLabel = new JLabel();
iconLabel.setIcon(null);
```

```
    // frame layout
    mainPanel.setLayout(null);
    mainPanel.add(iconLabel);
    mainPanel.add(buttonPanel2);

    // Set boundary
    iconLabel.setBounds(0, 0, 380, 315);
    buttonPanel2.setBounds(0, 310, 380, 50);

    // Add main panel into JFrame
    f.getContentPane().add(mainPanel,
       BorderLayout.CENTER);
    f.setSize(new Dimension(390, 400));

    // init timer for video
    timer = new Timer(20, new timerListener()); //
       create object timer
    timer.setInitialDelay(0);
    timer.setCoalesce(true);

    //allocate enough memory for the buffer used to
      receive data from the server
    buf = new byte[15000];

} // end constructor

//-------------------------------------------
//
//             Public Methods:
//
//-------------------------------------------

/**
 * Runs a thread, it has to be run as a thread in order
   to achieve the GUI.
 * There are two types of the protocols; TCP and UDP,
   depending on the
 * connectionCase. When a physical lost occurs, in
   order to achieve
 * the persistent data sessions, the client will ask
   the server to send
 * data again by using the previous parameters. It
   seems to be non-persistent
 * due to the new establishment of the connection but
   the idea of persistent
 * connection will be used instead.
 */
```

```
public void run() {

  if (connectionCase == 1)

    TCPStart();

  else

    UDPStart();

} // end run()

/**
 * Method to close all socket variables
 */
public void close() {

    try {

        if (br != null) {
            br.close();
        } // end if

        if (bw != null) {
            bw.close();
        } // end if

        if (socket != null) {
            socket.close();
        } // end if

    } catch (java.io.IOException io) {

        JOptionPane.showMessageDialog(this, "Input /
           Output error occured, you should restart",
           "Socket closed error",
           JOptionPane.INFORMATION_MESSAGE);

    } // end try - catch

} // end close()

/**
 * Method to open all socket variables
 *
 * @return boolean true if the connection can be
    established otherwise false
```

```
 */
public boolean open() {

    try {

        System.out.println("Trying to connect to the
           server....");
        /** use for both TCP request and RTSP request
             (for later UDP)
         * automatically connect to the host(server) */
        socket = new Socket(host, port); // ==>
        System.out.println("After create socket,
           connection ==> " + socket.isConnected());
        connect = true;

        // initialize buffer for both input and output
           streams, use socket for initialization
        br = new BufferedReader(new
          InputStreamReader(socket.getInputStream()));
        bw = new BufferedWriter(new
          OutputStreamWriter(socket.getOutputStream()));

    } catch (UnknownHostException uhe) {

        JOptionPane.showMessageDialog(this, "Unknown
           server, check the address");
        return false;

    } catch (IOException ioe) {

        JOptionPane.showMessageDialog(this, "Cannot
            connect to the server, server may be down or
            cable unplugged.", "Socket Error!",
            JOptionPane.INFORMATION_MESSAGE);

        return false;

    } // end try - catch

    return true; // if success

} // end open()

/**
 * Method to send the message to the server
 *
 * @param message - the string request
```

```
     */
    public void send(String message) {

        try {

            // write the message to the server using buffer
               writer
            bw.write(message);
            bw.newLine();
            bw.flush();

        } catch (SocketException se) {

            // true if the socket successfully connected to
               the server
            if (socket.isConnected()) {
                JOptionPane.showMessageDialog(this, "The
                    connection still established!");
            } else {
                JOptionPane.showMessageDialog(this, "Other
                    problems!");
            } // end if - else

        } catch (Exception e) {

            JOptionPane.showMessageDialog(this,
              "In send,  Check if connected to server",
              "Sending Error!", JOptionPane.ERROR_MESSAGE);

            close();

        }  // end try - catch

    } // end send()

    /**
     * Method to send the request to the server
     *
     * @param request_type - the UDP request
     */
    public void send_RTSP_request(String request_type) {

        try {

            // write the message to the server using buffer
               writer
```

```
        bw.write(request_type + " Cseq: " + RTSPSeqNb +
            CRLF);
        bw.flush();

    } catch (Exception ex) {

        System.out.println("in send Exception caught: "
            + ex);
        System.exit(0);

    } // end try - catch

} // end send_RTSP_request()

/**
 * Method to set the parameters after getting new
   connection
 *
 * @param socket - the TCP or RTSP communication socket
 * @param br - buffer for input stream
 * @param bw - buffer for output stream
 */
public void setParameters(Socket socket, BufferedReader
       br, BufferedWriter bw) {

    this.socket = socket;
    this.br = br;
    this.bw = bw;

} // end setParameters

/**
 * Method to retrieve the variable host
 *
 * @return host - the IP address of the host
 */
public String getHost() {

    return host;

} // end getHost()

/**
 * Method to retrieve the constant RTP_RCV_PORT
 *
 * @return RTP_RCV_PORT - the RTP destination port
 */
```

```
public int getUDPPort() {

    return (RTP_RCV_PORT);

} // end getUDPPort()

/**
 * Method to get the random RTP port
 *
 * @return - random RTP port number
 */
public int randomRTPPort() {

  return ((int) (Math.random() * 10000));

} // end randomRTPPort()

/**
 * Method to set the random RTP port
 *
 * @param newPort new RTP port numner
 */
public void setRandomRTPPort(int newPort) {

  newRTPPort = newPort;

} // end setRandomRTPPort()

/**
 * Method to get the random RTP port
 *
 * @param RTPsocket - new RTP socket
 */
public void setRTPSocket(DatagramSocket RTPsocket) {

  this.RTPsocket = RTPsocket;
  System.out.println("New RTP socket set!");

} // end setRTPSocket()


//----------------------------------------------
//
//              Private Methods
//
//----------------------------------------------
```

```
/**
 * Connect to the server either TCP or UDP
 * This client will not be terminated unless
 * the appropriate button will be pressed.
 *
 * Five steps to communicate with the client are
 *   Step 1 - Set up a client socket to send request
     to the server
 *   Step 2 - Set up the control agent
 *   Step 3 - Open appropriate streams for desired
     data exchange
 *   Step 4 - Communicate with the server via streams
 *   Step 5 - Close the opened socketconnection
 */

/**
 * Persistent TCP connection
 */
 private void TCPStart() {

     // variable initialization
     index = 0;
     String text = "";
     boolean done = false;

     if ((knowSize == false) && (connect == false)) {

         open(); // open the port for the TCP
                    connection
         textArea.append("Connected\n");

         // get the size of the data in order to
            manipulate the progress bar
         fileSize = findSize();

     } // end if

     while (!done) {

         System.out.println("In while loop");
         String textOut = "/get " + index;
         System.out.println("Command --> " +
            textOut);

         System.out.println("Request to server
             sent!");
```

```
send(textOut); // send the request to the
                  server

int spaceCounter = 0;
char[] testChar = new char[4];

// receive the content of the file until EOF
while (!socket.isClosed()) {

    try {

        try {

          socket.setSoTimeout(SOCKET_TIMEOUT
          ); // set timeout for the TCP
          socket

        } catch (SocketException se) {

         JOptionPane.showMessageDialog(this,
           "The connection has lost, error
           in the underlying protocol",
           "Timeout!",
           JOptionPane.ERROR_MESSAGE);

         JOptionPane.showMessageDialog(this,
           "The next job is to reestalish
           the session", "Next Process",
           JOptionPane.INFORMATION_MESSAGE);

         close(); // close all sockets
            before doing reconnection

         // if the connection is lost by
         timeout, it will reconnect
         automatically

       controlAgent.reconnectProcess("TCP");

        } // end try - catch

        text = br.readLine(); // read the
           incoming response from the server
        System.out.println("Received ==> " +
           text + " , index = " + index);
```

```
// very important for retrieving the
   rest of the data
controlAgent.setIndex(index);

if (text.equalsIgnoreCase("")) {

  System.out.println("spaceCounter =
    " + spaceCounter);
  spaceCounter++;

  if (spaceCounter == 3) {

    textArea2.append("\n");
    spaceCounter = 0;

  } // end if

} else{

  textArea2.append(text);
  spaceCounter = 0;

} // end if - else

if (text != null) {

    index++; // increment the
                counter
    progressBar.setValue(index); //
            set the progress bar

    // End of file
    if ((index) == fileSize) {

       textArea2.append("\n");

       JOptionPane.showMessageDialog
       (this, "End of File!");

       done = true; // set the logic
            to exit the outer loop
       break; // exit the inner loop

    } // end if

} // end if
```

```
} catch (NoRouteToHostException nrth) {

    JOptionPane.showMessageDialog(this,
       "The route to host!");
    JOptionPane.showMessageDialog(this,
       "The next job is to reestalish
       the session");
} catch (ConnectException ce) {

    JOptionPane.showMessageDialog(this,
       "The connection was refused
       remotely");
    JOptionPane.showMessageDialog(this,
       "The next job is to reestalish
       the session");
} catch (SocketException se) {

    System.out.println("The error is
       ===> " + se.getMessage());

    close(); // close the socket and I/O
        streams to quit inner loop

    JOptionPane.showMessageDialog(this,
        "The connection has lost, error
        in the underlying protocol",
        "Socket closed!",
        JOptionPane.ERROR_MESSAGE);
    JOptionPane.showMessageDialog(this,
        "The next job is to reestalish
        the session", "Next Process",
        JOptionPane.INFORMATION_MESSAGE);

    /**
     * After the physical connection is
       lost,
     * the client socket will be closed
       and the agent
     * is trying to establish the new
       commnication
     *
     * <p>
     * Five steps to communicate with
       the client are
     *   Step 1 - Close the opened
         socket
```

```
                 *   Step 2 - Call agent to do the
                     reconnect process
                 *   Step 3 - Do the iteration until
                     the connection is resumed
                 *   Step 4 - Continue communicate
                     with the server via streams
                 *   Step 5 - Get the rest of the
                     data until the end of the file
                 */

                // type of protocol should be passed
               controlAgent.reconnectProcess("TCP");

            } catch (IOException io) {

            } // end try - catch

        } // end inner while

    } // end outer while

    textArea.append("\n");

   textArea.append(String.valueOf(socket.isClosed())
   );
    JOptionPane.showMessageDialog(this, "End of
       File!");

} // end TCPStart()

/**
 * Persistent UDP connection
 */
private void UDPStart() {

    System.out.println("Random port = " +
       randomRTPPort());

    try {

        if ((knowSize == false) && (connect ==
               false)) {

            open();
            textArea.append("Connected\n");
            fileSize = findSize();
```

```
        } // end if

        /** construct a new DatagramSocket to
            receive RTP packets
         * from the server, on port RTP_RCV_PORT */
        RTPsocket = new
           DatagramSocket(RTP_RCV_PORT);

    } catch (SocketException se) {

        timer.stop();

        System.out.println("Socket exception: " +
            se);
        JOptionPane.showMessageDialog(this, "The
            connection has lost, error in the
            underlying protocol");

    } // end try - catch

    ready = setupUDPSession();

    // add text onto the GUI panel
    textArea.append("\n");
    textArea.append("Now waiting for the action from
       the user...\n");

} // end UDPStart()

/**
 * UDP session initialization
 */
private boolean setupUDPSession() {

  //init RTSP sequence number
  RTSPSeqNb = 1;

  /** Send SETUP message to the server to start the
      video then wait for listener for the next
      action */
  send("/Setup " + RTSPSeqNb + " " + RTP_RCV_PORT);

  return true;

} // end setupUDPSession()
```

```
/**
 * Method to control the physical connection
 *
 * @param client - the string request
 * @param port - port for TCP connection
 * @param host - IP address of the server
 */
private void fileControl(Client client, int port,
    String host) {

    controlAgent = new ConnectionControl(client,
        port, host);

} // end fileControl()

/**
 * Method to find the size of the file
 *
 * @return size the size of the file in bytes
 */
private int findSize() {

    int size = 0;

    try {

      open();

      String text = new String("");
      send("/size");
      textArea.append("Now getting the file size from
          the server...");
      text = br.readLine();
      textArea.append(" ---> " + text + " bytes\n");

      if (knowSize == false) {

        size = Integer.parseInt(text);
        knowSize = true;

      } // end if

      progressBar.setMaximum(size - 3);

    } catch (NumberFormatException ex) {
        System.out.println(ex.getMessage());
    } catch (IOException ioex) {
```

```
        System.out.println(ioex.getMessage());
    } catch (NullPointerException npe) {
        System.out.println(npe.getMessage());
    }

    return size;

} // end findSize()

/**
 * Method to parse the request from the server
 */
public void parse_server_response() {

  try {

    System.out.println("in parse_server_response(),
        waiting for response...");
    textArea.append("\nin parse_server_response(),
        waiting for response...");
    //parse request line and extract the
      request_type:
    String requestLine = br.readLine();

    System.out.println("RTSP Client - Received from
        Server:");
    System.out.println("Received --> " +
        requestLine);
    textArea.append("\nReceived --> " + requestLine);

    if (requestLine.equalsIgnoreCase("EOF")) {

      JOptionPane.showMessageDialog(this, "End of the
          video");
      ready = false;

      int reply = JOptionPane.showConfirmDialog(null,
           "Would like to see again?" , "Ask for your
           permission", JOptionPane.YES_NO_OPTION);
      if (reply == JOptionPane.YES_OPTION)
        JOptionPane.showMessageDialog(this, "Please
            press Play button to sees it again");
      else {
        JOptionPane.showMessageDialog(this, "Return
            to main menu");
        f.setVisible(false);
      }
```

```
      } // end if - else if

    } catch (Exception ex) {

      System.out.println("Exception caught: " + ex);
      System.exit(0);

    } // end try - catch

  } // end parse_RTSP_request()


//-------------------------------------------------
//
//                  Main Method
//
//-------------------------------------------------

/**
 * Main method to start the client
 *
 * @param arg argument list
 */
public static void main(String arg[]) {

    Client client;

    if (arg.length != 2) {
        System.out.println("Usage: java Client
            <hostname> <portnumber>");
        System.exit( -1);
    }

    client = new Client(arg[0],
        Integer.parseInt(arg[1]));

    // exits when the window is closed
    client.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {

            System.exit( -1);
        }
    }
    );

} // end main()
```

```
//------------ I N N E R   C L A S S -------------//

//------------------------------------------------
//
//          Action Listener Methods:
//
//------------------------------------------------

private class MenuHandler implements ActionListener {

    public void actionPerformed(ActionEvent e) {

        if (e.getSource().equals(size)) {

            open();
            fileSize = findSize();

        } else if
              (e.getSource().equals(filePersistent)) {

            connectionCase = 1;

            textArea.append("\nTCP connection
               selected!\n");
            textArea.append(Please press connect button
               to receive the data...\n");

        } else if
              (e.getSource().equals(filePersistent2)) {

            // do UDP
            connectionCase = 2;
            UDPThread = new Thread(Client.this);

            textArea.append("\nUDP connection
               selected!\n");
            textArea.append("Please press connect
               button to receive the data...\n");

        } // end if - else if

    } // end actionPerform()

} // end class MenuHandler
```

```
//--------------------------------------------
//   Handler for Close and Connect buttons
//--------------------------------------------

private class ButtonHandler extends JFrame implements
              ActionListener {

    public void actionPerformed(ActionEvent e) {

        if (e.getSource().equals(buttonClose)) {

            System.exit(0);

        } else if (e.getSource().equals(buttonClose2)){

            RTSPSeqNb++;

            //Send CLOSE message to the server
            send_RTSP_request("CLOSE");
            textArea.append("\nClient Process Close
                button pressed!\n");

            timer.stop();
            f.setVisible(false); // disable the
                                 visibility of the GUI

        }else if (e.getSource().equals(buttonConnect)){

            textArea.append("\nButton Connect
                pressed!\n");

            if (connectionCase == 0) {

              JOptionPane.showMessageDialog(this,
                   "Please choose one of the options
                   before hitting the connect button",
                   "Button Error!",
                   JOptionPane.ERROR_MESSAGE);

            } else if (connectionCase == 1) {

                k.setVisible(true); // Set the JFrame
                                       visible
                findSize();

            } else {
```

```
            f.setVisible(true); // Set the JFrame
                                   visible
            findSize();

            UDPThread.start();// start the UDP
                                 persistent process

        } // end if - else

    } else if (e.getSource().equals(buttonClose3)){

        textArea.append("\nTCP Close button
           pressed!\n");
        TCPThread.suspend();

        int reply =
            JOptionPane.showConfirmDialog(this, "Do
            you really want to close?", "Ask for
            confirmation",
            JOptionPane.YES_NO_OPTION);

        if (reply == JOptionPane.YES_OPTION) {

          TCPThread.stop();
          textArea2.setText("");
          k.setVisible(false); // disable the
                                 visibility of the GUI

        } else {

          TCPThread.resume();

        }

    } else if (e.getSource().equals(buttonData)) {

        textArea.append("\nNow receiving the data
            from the server...\n");

        TCPThread = new Thread(Client.this);
        TCPThread.start();// start the TCP
                             persistent process

    } // end if - else if

} // end actionPerform()
```

```
} // end class ButtonHandler

//------------------------------
//   Handler for Play button
//------------------------------

private class playButtonListener implements
         ActionListener {

    public void actionPerformed(ActionEvent e) {

        System.out.println("Play Button pressed !");
        textArea.append("\n");
        textArea.append("Play Button pressed !");
        textArea.append("\n");

        if (ready == false) {

          ready = setupUDPSession();

        } // end if

        //increase RTSP sequence number
        RTSPSeqNb++;

        //Send PLAY message to the server
        send_RTSP_request("PLAY");
        textArea.append("Now receiving the video from
            the server...\n");

        //state = PLAYING;
        System.out.println("New RTSP state:PLAYING");

        //start the timer
        timer.start();

    } // end actionPerformed()

} // end inner class playButtonListener

//------------------------------
//  Handler for Pause button
//------------------------------

class pauseButtonListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
```

```
        System.out.println("Pause Button pressed !");
        textArea.append("\n");
        textArea.append("Pause Button pressed !");
        textArea.append("\n");

        //increase RTSP sequence number
        RTSPSeqNb++;

        //Send TEARDOWN message to the server
        send_RTSP_request("PAUSE");
        textArea.append("The video will be paused and
            wait for the action...\n");

        //stop the timer
        timer.stop();

    } // end actionPerformed()

} // end inner class tearButtonListener

//-------------------------------------
//        Handler for timer
//-------------------------------------

class timerListener extends JFrame implements
        ActionListener {

    public void actionPerformed(ActionEvent ev) {

        //Construct a DatagramPacket to receive data
          from the UDP socket
        rcvdp = new DatagramPacket(buf, buf.length);

        try {

            // Set TimeOut value of the socket.
            RTPsocket.setSoTimeout(SOCKET_TIMEOUT);

            //receive the DP from the socket:
            RTPsocket.receive(rcvdp);

            //create an RTPpacket object from the DP
            RTPpacket rtp_packet = new
               RTPpacket(rcvdp.getData(),
               rcvdp.getLength());
```

```
      //print important header fields of the RTP
        packet received:
      System.out.println("Got RTP packet with
          SeqNum # " +
          rtp_packet.getsequencenumber() + "
          TimeStamp " + rtp_packet.gettimestamp()
          + " ms, of type " +
          rtp_packet.getpayloadtype());

      index = rtp_packet.getsequencenumber();
      controlAgent.setIndex(index); // set the
          reference of the video

      //get the payload bitstream from the
        RTPpacket object
      int payload_length =
         rtp_packet.getpayload_length();
      byte[] payload = new byte[payload_length];

      rtp_packet.getpayload(payload);

      //get an Image object from the payload
        bitstream
      Toolkit toolkit =
         Toolkit.getDefaultToolkit();
      Image image = toolkit.createImage(payload,
         0, payload_length);

      //display the image as an ImageIcon object
      icon = new ImageIcon(image);
      iconLabel.setIcon(icon);

  } catch (InterruptedIOException iioe) {

    if (ready == true ) {
      System.out.println("In timer
          InterruptIOException caught: " +
          iioe.getMessage());
      System.out.println("Seq num = " + index);
      timer.stop();
      System.out.println(timer.isRunning());

      close(); // close the socket and I/O
                  streams to quit inner loop

      JOptionPane.showMessageDialog(this, "The
          connection has lost, error in the
```

```
                underlying protocol", "Socket closed!",
                JOptionPane.ERROR_MESSAGE);
            JOptionPane.showMessageDialog(this, "The
                next job is to reestalish the session",
                "Next Process",
                JOptionPane.INFORMATION_MESSAGE);

            /**
             * After the physical connection is lost,
             * the client socket will be closed and the
               agent
             * is trying to establish the new
               commnication
             *
             * <p>
             * Five steps to communicate with the
               client are
             *   Step 1 - Close the opened socket
             *   Step 2 - Call agent to do the
                 reconnect process
             *   Step 3 - Do the iteration until the
                 connection is resumed
             *   Step 4 - Continue communicate with the
                 server via streams
             *   Step 5 - Get the rest of the data
                 until the end of the file
             */

            // type of protocol should be passed on
            controlAgent.reconnectProcess("UDP");
          }
        } catch (IOException ioe) {
            System.out.println(" in timer IOException
               caught: " + ioe.getMessage());

        } catch (Exception e) {
            System.out.println(" in timer Exception
                caught: " + e.getMessage());

        } // end try - catch

    } // end actionPerformed()

  } // end inner class timerListener

} // end class Client
```

```
/**
 * Title: API Development for Persistent Data Sessions
          Support
 * Description: Persistency API class
 * Compiler : JBuilder 9
 * Author CPT.Chayutra Pailom THA
 * Date : January 20, 2005
 */

import java.io.*;
import java.net.*;
import java.awt.*;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This is the critical part of the project. This class is
   intended to be an API
 * to reconnect and send a request instead of the real
   client. It will be activated
 * after the physical connection is lost and it will try to
   detect and reestablish
 * the new connection between the client and the server.
   Finally if the connection
 * is resumed, it will return the parameters and the rest
   of the process back to
 * the real client. This class is intended to be a
   universal function for the
 * reconnection. The type of protocol,'TCP' or 'UDP', is
   required to be passed
 * through this function inorder to do the different job
   for sending the request.
 *
 *
 * Expected method: reconnectProcess() - it will do the
   iteration forever unless
 *  the physical connection is resumed
 *
 * @author CPT Chayutra Pailom THA
 */
public class ConnectionControl {
```

```java
//-------------------------------------------------------
//
//                     Data Members:
//
//-------------------------------------------------------

/** The client object of the project */
private Client client;

/** Socket for TCP and RTSP request */
private Socket socket;

/** Port to communicate with the server */
private int port;

/** The IP address of the server */
private String host;

/** The buffer for input stream */
private BufferedReader br;

/** The buffer for output stream */
private BufferedWriter bw;

/** The reference of the file for TCP and UDP before the
    lost connection */
private int index;

/** Logic to control the iteration */
private boolean done;

/** socket to be used to send and receive UDP packets */
private static DatagramSocket RTPsocket;

//--------------------------------------------------------
//
//                        Constructor:
//
//--------------------------------------------------------

/**
 * Default constructor
 *
 * @param client - client object
 * @param port - port number for TCP, RTSP communication
 * @param host - IP address of the server
 */
```

```
public ConnectionControl(Client client, int port, String
       host) {

  this.client = client;
  this.port = port;
  this.host = host;

  initialization();

} // end constructor

//-----------------------------------------
//
//             Public Methods:
//
//-----------------------------------------

/**
 * Method to initialize the important parameters
 */
public void initialization() {

  index = 0; //very important to retrieve the content of
    the file
  done = false; // variable for the iteration for doing
    the new connection

} // end initializeation()

/**
 * Method to reconnect the communication between the
  client and the server
 *
 * @param type - type of protocol to be passed on this
               fuction
 */
public void reconnectProcess(String type) {

  System.out.println("Trying to establish the
    connection...");
  System.out.println("Index to read file = " + index);
  client.buttonData.setText("Pause");

  /**
   * Reconnect to the server either TCP or UDP
   * This object is trying to help the client to detect
```

```
 * the physical connection. If the connection is
   resumed
 * it will send the request either TCP or RTSP for the
 * client and then return the value to the client. The
   client
 * will continue its job until the end of the file.
 *
 * Five steps to reconnect the communication with the
   server are
 *   Step 1 - Trying to establish the connection via
     the socket
 *   Step 2 - If it is resumed, open appropriate
     streams for desired data exchange. If it is not,
     loop forever
 *   Step 3 - Communicate with the server via streams
     by asking and sending the new request to the server
     using 'index' as an offset of the retrieved file.
 *   Step 4 - Set the parameters back to the client
     (socket, br bw)
 *   Step 5 - Return the process back to the client
     (br.readLine())
 */

while (!done) {

    try {

      /** create new socket, automatically connected if
          the physical
       *  connection resumed
       */
       socket = new Socket(host, port);
       JOptionPane.showMessageDialog(client,
         "Connection resumed!!!", "Client Status",
         JOptionPane.INFORMATION_MESSAGE);

       // if success, it will print. If not it will go
          to the exception
       System.out.println("After create socket,
         connection ==> " + socket.isConnected());
       open();

       // set the new parameters for the client
       client.setParameters(socket, br, bw);

    if (type.equalsIgnoreCase("TCP")) {
```

```
        System.out.println("Type = " + type);
        sendTCPRequest();

      } else {

        try {

          System.out.println("Type = " + type);
          sendUDPRequest();

        } catch (Exception e) {}

      } // end if - else

      done = true; // set the exit of the loop

    } catch (UnknownHostException uhe) {

      JOptionPane.showMessageDialog(client, "Unknown
        server, check the address");
    } catch (IOException ioe) {

      System.out.println("No connection resumed. Still
        trying to connect to the server.....");
      try {

        Thread.sleep(200);

      } catch (Exception e) {}

    } // end try - catch

  } //end while

} // end reconnectProcess()

/**
 * Method to set the reference of the file
 *
 * @param index - the offset the file
 */
public void setIndex(int index) {

  this.index = index;

} // end setIndex()
```

```
//-------------------------------------------
//
//              Private Methods
//
//-------------------------------------------

/**
 * Method to open all socket variables
 */
private void open() {

  try {

    br = new BufferedReader(new
      InputStreamReader(socket.getInputStream()));
    bw = new BufferedWriter(new
      OutputStreamWriter(socket.getOutputStream()));

  } catch (UnknownHostException uhe) {

    JOptionPane.showMessageDialog(client, "Unknown
      server, check the address");

  } catch (IOException ioe) {

    JOptionPane.showMessageDialog(client, "Cannot connect
      to the server, server may be down");

  } // end try - catch

} // end open()

/**
 * Method to send TCP request to the server instead of
   the client
 */
private void sendTCPRequest() {

  String textOut = "/get " + index;
  System.out.println("Command --> " + textOut);
  JOptionPane.showMessageDialog(client, "The transmission
    is continuing from the point it lost!", "Client
    Status", JOptionPane.INFORMATION_MESSAGE);

  client.buttonData.setText("Get Data");
  client.send(textOut);
```

```
  } // end sendUDPRequest()

  /**
   * Method to send UDP request to the server instead of
     the client
   *
   * @throws Exception
   */
  private void sendUDPRequest() throws Exception {

    JOptionPane.showMessageDialog(client, "The video is
      continuing from the point it lost!", "Client Status",
      JOptionPane.INFORMATION_MESSAGE);

    int newRTPPort = client.randomRTPPort();
    System.out.println("New port = " + newRTPPort);

    /** construct a new DatagramSocket to receive RTP
        packets from the server, on port RTP_RCV_PORT */
    RTPsocket = new DatagramSocket(newRTPPort);

    client.setRTPSocket(RTPsocket);
    client.setRandomRTPPort(newRTPPort);

    // send the requests to get the video
    client.send("/Setup " + index + " " + newRTPPort);
    client.send_RTSP_request("PLAY");

    client.textArea.append("\nSetup and Play requests
      sent!\n");

    client.timer.start(); // continue the timer after
      sending the new request

    client.parse_server_response();

  } // end sendRequest()

} // end class ConnectionControl
```

```
/**
 *  Title:  API  Development  for  Persistent  Data  Sessions
    Support
 * Description: RTP packets for video file transfer
 * Compiler : JBuilder 9
 * Author CPT.Chayutra Pailom THA
 * Date : January 20, 2005
 */

public class RTPpacket{

//-----------------------------------------
//
//           Data Members:
//
//-----------------------------------------

  /** size of the RTP header: */
  static int HEADER_SIZE = 12;

  /** Version fields the RTP header */
  public int Version;

  /** Padding field */
  public int Padding;

  /** Extension field */
  public int Extension;

  /** Contributing source */
  public int CC;

  /** Marker field */
  public int Marker;

  /** Payload of the RTP packet */
  public int PayloadType;

  /** Sequence number of the RTP packet */
  public int SequenceNumber;

  /** Timestamp */
  public int TimeStamp;

  /** Synchronization source */
  public int Ssrc;
```

```
  /** Bitstream of the RTP header */
  public byte[] header;

  /** Size of the RTP payload */
  public int payload_size;

  /** Bitstream of the RTP payload */
  public byte[] payload;

//--------------------------------------------
//
//              Constructor:
//
//--------------------------------------------

  /**
   * Set an RTPpacket object from header fields and payload
     bitstream
   *
   * @param PType the type of the payload
   * @param Framenb the sequence number
   * @param Time time stamp
   * @param data the array of byte of the data
   * @param data_length the length of the data
   */
  public RTPpacket(int PType, int Framenb, int Time, byte[]
    data, int data_length){

    // Fill by default header fields:
    Version = 2;
    Padding = 0;
    Extension = 0;
    CC = 0;
    Marker = 0;
    Ssrc = 0;

    // Fill changing header fields:
    SequenceNumber = Framenb;
    TimeStamp = Time;
    PayloadType = PType;

    // Build the header bistream:
    header = new byte[HEADER_SIZE];

    // RTP header
    header[0] = (byte) (header[0] | Version << 7);
    header[0] = (byte) (header[0] | Padding << 5);
```

```
    header[0] = (byte) (header[0] | Extension << 4);
    header[0] = (byte) (header[0] | CC << 3);

    header[1] = (byte) (header[1] | Marker << 7);
    header[1] = (byte) (header[1] | PayloadType << 6);

    // Sequence number
    header[2] = (byte) (SequenceNumber >> 8);
    header[3] = (byte) (SequenceNumber & 0xFF);

    // Timestamp , all 32 bits
    header[4] = (byte) (TimeStamp >> 24);
    header[5] = (byte) (TimeStamp >> 16);
    header[6] = (byte) (TimeStamp >> 8);
    header[7] = (byte) (TimeStamp & 0xFF);

    // Synchronization source, all 32 bits
    header[8] = (byte) (Ssrc >> 24);
    header[9] = (byte) (Ssrc >> 16);
    header[10] = (byte) (Ssrc >> 8);
    header[11] = (byte) (Ssrc & 0xFF);

    // Fill the payload bitstream:
    //--------------------------
    payload_size = data_length;
    payload = new byte[data_length];

    // Fill payload array of byte from data (given in
       parameter of the constructor)
    payload = data;

  } // end constructor

  /**
   * Set an RTPpacket object from the packet bistream
   *
   * @param packet the header of the bitstream
   * @param packet_size the total packet size
   */
  public RTPpacket(byte[] packet, int packet_size) {

    // Fill default fields:
    Version = 2;
    Padding = 0;
    Extension = 0;
    CC = 0;
    Marker = 0;
```

```
  Ssrc = 0;

  // Check if total packet size is lower than the header
     size
  if (packet_size >= HEADER_SIZE) {

    // Get the header bitsream:
    header = new byte[HEADER_SIZE];
    for (int i = 0; i < HEADER_SIZE; i++)
      header[i] = packet[i];

    // Get the payload bitstream:
    payload_size = packet_size - HEADER_SIZE;
    payload = new byte[payload_size];

    for (int i = HEADER_SIZE; i < packet_size; i++)
      payload[i - HEADER_SIZE] = packet[i];

    // Interpret the changing fields of the header:
    PayloadType = header[1] & 127;
    SequenceNumber = unsigned_int(header[3]) + 256 *
      unsigned_int(header[2]);
    TimeStamp = unsigned_int(header[7]) + 256 *
      unsigned_int(header[6]) + 65536 *
      unsigned_int(header[5]) + 16777216 *
      unsigned_int(header[4]);

  } // end if

} // end constructor

/**
 * Get payload
 *
 * @param data - the data of the payload
 *
 * @return the - the payload bistream of the RTPpacket
   and its size
 */
public int getpayload(byte[] data) {

  for (int i = 0; i < payload_size; i++)
    data[i] = payload[i];

  return(payload_size);

} // end getpayload()
```

```
/**
 * Get length of the payload
 *
 * @return the length of the payload
 */
public int getpayload_length() {

  return(payload_size);

} // end getpayload_length()

/**
 * Get total length of the packet
 *
 * @return the length of the packet
 */
public int getlength() {

  return(payload_size + HEADER_SIZE);

} // end getlength()

/**
 * Get the packet size
 *
 * @param packet the array of byte of the packet
 *
 * @return the total size of the packet
 */
public int getpacket(byte[] packet) {

  // Construct the packet = header + payload
  for (int i = 0; i < HEADER_SIZE; i++)
      packet[i] = header[i];

  for (int i = 0; i < payload_size; i++)
      packet[i + HEADER_SIZE] = payload[i];

  //return total size of the packet
  return(payload_size + HEADER_SIZE);

} // end getpacket()

/**
 * Get the timestamp
 *
```

```
   * @return the value of the timestamp
   */
  public int gettimestamp() {

    return TimeStamp;

  } // end gettimestamp()

  /**
   * Get the sequence number
   *
   * @return the sequence number
   */
  public int getsequencenumber() {

    return SequenceNumber;

  } // end getsequencenumber()

  /**
   * Getpayloadtype
   *
   * @return the payload type
   */
  public int getpayloadtype() {

    return PayloadType;

  } // end getpayloadtype()

  /**
   * Check and return the proper unsigned number
   *
   * @param nb - an unsign bit
   *
   * @return the unsigned value of 8-bit integer nb
   */
  static int unsigned_int(int nb) {

    if (nb >= 0)
      return(nb);
    else
      return (256 + nb);

  } // end unsigned_int()

} // end class RTPpacket
```

```
/**
 * Title: API Development for Persistent Data Sessions
   Support
 * Description: Application server
 * Compiler : JBuilder 9
 * Author CPT.Chayutra Pailom THA
 * Date : January 20, 2005
 */

import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This is the server of the project. This server is
   expected to coordinate
 * with one client at a time. It will create the thread as
   an 'agent' of the
 * server. This class is intended to use socket programming
   for both types of
 * protocols and graphic user interface on order to show
   the process.
 *
 * Expected number of clients which server can handle: 30
 *
 * @author CPT Chayutra Pailom THA
 */
public class Server extends JFrame implements
   ActionListener {

  //-------------------------------------------------------
  //
  //                Data Members:
  //
  //-------------------------------------------------------

  //--------- GUI for indicating server process ---------/

  /** JFrame of the server object */
  public JFrame g, h;

  /** Lebel for showing the server process */
  public JLabel label, label2;
```

```
/** The area to show the process of the server */
private JTextArea textArea;

/** The button to shutdown the visibility of the GUI */
private JButton buttonExit;

//---------- Server socket variables ------------/

/** Server socket to be used to wait for the client
    connection */
private ServerSocket serverSocket;

/** Socket to be used to send the TCP and RTSP request */
private Socket clientSocket;

/** Array of server agent talking to the client */
private ServerThread clientThread[];

/** Number of created socket */
private int socketNumber;

/** Input stream filters */
private BufferedReader br;

/** Output stream filters */
private BufferedWriter bw;

/** Port to communicate with the server */
private int port;

/** Buffer used to store the file content to send to the
   client */
private char[] buffer;

/** Size of the file in bytes */
private int fileSize;

/** File to be sent to the client */
private String fileName;

/** File input stream to be read */
private FileInputStream fis;

/** Status of the connection with the client */
private boolean connectionLost;
```

```
/** Expected number of clients */
private int noOfClients;

/** Sequence of client connected to the server */
private int clientNumber;

//-----------------------------------------------------
//
//                Constructor:
//
//-----------------------------------------------------

/**
 * Default constructor
 *
 * @param name - file name to be retrieved
 * @param no - port number of the server to be connected
 */
public Server(String name, int no) {

  // initialize variables
  fileName = name;
  port = no;
  connectionLost = false;
  clientNumber = 0;
  noOfClients = 0;

  // initialize agents
  clientThread = new ServerThread[30]; // can handle for
                                         30 threads
  //Socket[] clientSocket = new Socket[30];

  // frame specifications
  Container container = getContentPane();
  this.setTitle("Server Process");
  this.setSize(500, 300);
  this.setLocation(0, 0);

  // buttons and textcomponents
  buttonExit = new JButton("Exit");
  buttonExit.addActionListener(this);
  textArea = new JTextArea();
  textArea.setEditable(false);
  JScrollPane scrollPane = new JScrollPane(textArea);

  // border layout placements
  container.setLayout(new BorderLayout());
```

```
container.add(scrollPane, "Center");
container.add(buttonExit, "South");

this.setVisible(true); // set visibiity of the main GUI
boolean done = false;

// GUI for TCP or UDP counter
g = new JFrame(" UDP Counter");
h = new JFrame("TCP Counter");

// add window listener
g.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
});
h.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
});

// initialize label
label = new JLabel("Send frame #        ",
   JLabel.CENTER);
label2 = new JLabel("Send character #        ",
   JLabel.CENTER);

// frame attributes both TCP and UDP
g.getContentPane().add(label, BorderLayout.CENTER);
g.setSize(150, 50);
g.setLocation(280, 50);
h.getContentPane().add(label2, BorderLayout.CENTER);
h.setSize(300, 100);
h.setLocation(250, 50);

// start the server
try {

  serverSocket = new ServerSocket(port);

  System.out.println("Binding to port : " + port +
                     ", please wait  ...");
  System.out.println("Server started : " +
    serverSocket);
  System.out.println("Server IP : " +
    serverSocket.getInetAddress());
```

```
    readFile(fileName); // read the content of the file
                           into buffer

    start(); // go start the server

  }
  catch (java.io.IOException ioe) {

    System.out.println("Cannot bind to port, port may be
      using by another application ");

  } // end try - catch

} // end constructor

//-------------------------------------------
//
//              Public Methods:
//
//-------------------------------------------

/**
 * Set the logic of the communication status
 *
 * @param connectionLost - the status of the
   communication
 */
public void setConnectionLost(boolean connectionLost) {

  this.connectionLost = connectionLost;

} // end setConnectionLost()

/**
 * Listener for the exit button
 *
 * @param e - Listener for the GUI
 */
public void actionPerformed(ActionEvent e) {

  if (e.getSource().equals(buttonExit)) {
    System.exit(0);
  }

} // end actionPerformed()
```

```
/**
 * Add the text to the GUI panel
 *
 * @param message - message to be added on the panel
 */
public void addTextArea(String message) {

  textArea.append(message);
  textArea.append("\n");

} // end addTextArea()


//-------------------------------------------
//
//            Private Methods
//
//-------------------------------------------

/**
 * Method to start the server
 */
private void start() {

  // do the iteration until it reach the maximum number
     of threads
  while (noOfClients < clientThread.length) {

    try {

      addTextArea("Server is waiting for a client to
        establish the connection...\n");

      // create the socket to communicate with the client
      clientSocket = serverSocket.accept();

      addTextArea("Server accepted");
      JOptionPane.showMessageDialog(this, "New connection
        is established!");

      addThread(clientSocket, clientNumber); // create
        the thread for communication

      addTextArea("Thread #" + (clientNumber + 1) + "
        created!\n");

      if (clientNumber > 0) {
```

```
        clientThread[clientNumber -
          1].setConnectionLost(true);
        removeThread(clientNumber - 1);

      } // end if

      clientNumber++; // increment the number of socket

    } catch (java.io.IOException ioe) {

      System.out.println("Client acceptance error ==> " +
         ioe.getMessage());

    } catch (Exception e) {

      System.out.println(e.getMessage());

    } // end try - catch

  } // end while

  // inform the user when it reaches the maximum number
     of clients
  if (noOfClients == clientThread.length) {

    JOptionPane.showMessageDialog(this, "Server can't add
       anymore threads!");
  }

} // end start()

/**
 * Method to read the content of the file
 *
 * @param fileName - file to be read into buffer
 */
private void readFile(String fileName) {

  try {

    boolean fileExist = true;
    File file = new File(fileName);

    try {

      fis = new FileInputStream(fileName);
```

```
    } catch (FileNotFoundException e) {

      fileExist = false;

    } // end try - catch

    if (fileExist) {

      System.out.println("File " + fileName + " is
         found");
      fileSize = (int) file.length(); // get the length
         of the file
      System.out.println("File Size is : " + fileSize + "
         bytes");

      // declare the array of character to be kept for
         sending to the client
      buffer = new char[fileSize];

      // buffer initialization
      FileReader fr = new FileReader(file);
      br = new BufferedReader(fr);

      br.read(buffer, 0, fileSize);  // read the file
                                         into buffer

    } else {

      System.out.println("file '" + fileName + "' can not
        be found");

      for (int i = 0; i < fileSize; i++) {

        buffer[i] = (char) i;

      } // end for

    } // end if - else

  } catch (Exception e2) {

    System.out.println("Problem in reading file" + e2 +
       "");

  } // end try - catch
```

```
} // end readFile()

/**
 * Method to add the new connection as a thread
 *
 * @param socket - socket for TCP or RTSP communication
 * @param clientNumber - the sequence number of connected
   client
 *
 * @throws Exception
 */
private void addThread(Socket socket, int clientNumber)
        throws Exception {

    clientThread[clientNumber] = new ServerThread(this,
       socket, fileName, (clientNumber + 1));
    open(socket); // open the streams

    // set important parameters
    clientThread[clientNumber].setParameters(br, bw,
       buffer, fileSize);
    clientThread[clientNumber].setConnectionLost(false);

    clientThread[clientNumber].start(); // start the
                                           thread

} // end thread

/**
 * Method to remove the expired connection
 *
 * @param clientNumber - the sequence number of connected
    client
 */
private void removeThread(int clientNumber) {

  ServerThread toTerminate = clientThread[clientNumber];
  textArea.append("Removing thread# " + (clientNumber +
     1) + "\n");

  try {

    //toTerminate.stop();
    toTerminate.close();

  } catch (IOException ioe) {
```

```
      textArea.append("Error in closing thread\n");

    } // end try - catch

    textArea.append("\nThread# " + (clientNumber + 1) + "
       removed!\n\n");

  } // end removeThread()

  /**
   * Method to open streams filters
   *
   * @param clientSocket - socket for the server - client
                             communication
   *
   * @throws IOException
   */
  private void open(Socket clientSocket) throws IOException
  {

      br = new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()));
      bw = new BufferedWriter(new
        OutputStreamWriter(clientSocket.getOutputStream()));

  } // end open()

  //-----------------------------------------------
  //
  //                    Main Method
  //
  //-----------------------------------------------

  public static void main(String arg[]) {

    try {

      Server server = new Server(arg[0],
        Integer.parseInt(arg[1]));

    }
    catch (NumberFormatException nfex) {

      System.out.println(nfex.getMessage());

    } catch (ArrayIndexOutOfBoundsException aiobe) {
```

```
      System.out.println("USage: Java Server
         <portnumber>\n");

    } catch (Exception e) {

      System.out.println(e.getMessage());

    } // end try - catch

  } // end main()

} // end class Server
```

```
/**
 * Title: API Development for Persistent Data Sessions
   Support
 * Description: Server agent
 * Compiler : JBuilder 9
 * Author CPT.Chayutra Pailom THA
 * Date : January 20, 2005
 */

import java.awt.event.*;
import java.net.*;
import java.io.*;
import javax.swing.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

/**
 * This class is expected to communicate with only one
   client per connection.
 * It acts as an 'agent' of the server. This class is
   intended to use socket
 * and important variables passed from the server for both
   types of protocols
 * and it uses graphic user interface on order to show the
   process.
 *
 * @author CPT Chayutra Pailom THA
 */
public class ServerThread extends Thread implements
   Runnable {

  /** Socket to be used to send the TCP and RTSP request */
  private Socket clientSocket;

  /** The main object of the execution */
  private Server server;

  /** Input stream filters */
  private BufferedReader br;

  /** Output stream filters */
  private BufferedWriter bw;

  /** Status of the connection with the client */
  private boolean connectionLost;
```

```
/** The buffer for containing the text file content */
private char[] buffer;

/** Size of the file in bytes */
private int fileSize;

/** File to be sent to the client */
private String fileName;

/** Offset for retriving the data */
private int indexToReadFile;

/** Status of the thread */
public boolean dead;

/** Client number */
private int ID;

//---------------- RTP Variables -------------------/

/** Socket to be used to send and receive UDP packets */
private DatagramSocket RTPsocket;

/** UDP packet containing the video frames */
private DatagramPacket senddp;

/** Client IP address */
private InetAddress ClientIPAddr;

/** destination port for RTP packets (given by the RTSP
    Client) */
private int RTP_dest_port = 0;

//--------------- VDO Variables -------------------/

/** Image nb of the image currently transmitted */
private int imagenb;

/** VideoStream object used to access video frames */
private VideoStream video;

/** RTP payload type for MJPEG video */
private static int MJPEG_TYPE = 26;

/** Frame period of the video to stream, in ms */
private static int FRAME_PERIOD = 100;
```

```
/** Length of the video in frames */
private static int VIDEO_LENGTH = 500;

/** Timer used to send the images at VDO frame rate */
private Timer timer;

/** Buffer used to store the images to send to client */
private byte[] buf;

/** Sequence number of RTSP messages within session */
private int RTSPSeqNb;

/** End of command */
private final static String CRLF = "\r\n";

/** rtsp states */
final static int INIT = 0;
final static int READY = 1;
final static int PLAYING = 2;

/** rtsp message types */
final static int PLAY = 3;
final static int PAUSE = 4;
final static int CLOSE = 5;

/** Logic for iteration control */
private boolean done;

//------------------------------------------------------
//
//                   Constructor:
//
//------------------------------------------------------

/**
 * Default constructor
 *
 * @param pServer - the server object
 * @param pSocket - socket for TCP or RTSP communication
 * @param fileName - file to be retrieved
 * @param threadID - ID of the thread
 *
 * @throws Exception
 */
public ServerThread(Server pServer, Socket pSocket,
    String fileName, int threadID) throws Exception {
```

```
    // variable assignment
    server = pServer;
    clientSocket = pSocket;
    this.fileName = fileName;
    ID = threadID;

    // initialize variables
    dead = false;
    imagenb = 0;
    RTSPSeqNb = 0;

    // init timer
    TimerHandler timerListener = new TimerHandler();
    timer = new Timer(FRAME_PERIOD, server);
    timer.addActionListener(timerListener);
    timer.setInitialDelay(0);
    timer.setCoalesce(true);

    //allocate memory for the sending buffer
    buf = new byte[15000];

  } // end constructor

  /**
   * Method to execute the runnable object
   */
  public void run() {

    try {

      // loop until the socket get closed
      while (!clientSocket.isClosed()) {

        //clientSocket.setSoTimeout(SOCKET_TIMEOUT);
        System.out.println(
            "In while loop and wait for the command...");

        // get the request from the client - one request
           per session
        String text = new String(br.readLine());
        System.out.println(text);

        if (text != null) {

          handle(text); // server activation

        } // end if
```

```
    // check for the end of the file
    if (((indexToReadFile) == buffer.length) ||
       (imagenb >= VIDEO_LENGTH)) {

      try {

        close(); // close when done, another thread is
                    waiting for new jobs

      } catch (java.io.IOException io) {

        System.out.println(io.getMessage());

      } // end try - catch

      break; // quit inner loop

    } // end if

  } // end while

} catch (SocketTimeoutException stoe) {

  System.out.println("SocketTimeoutException ==> " +
     stoe.getMessage());

} catch (SocketException se) {

  System.out.println("Socket Exception ==> " +
     se.getMessage());

  try {

    close();

  } catch (IOException ioe) {}

} catch (IOException ioe) {

  System.out.println("IOException ==> " +
     ioe.getMessage());

} catch (java.lang.NullPointerException npe) {

  System.out.println("NullPointerException ==> " +
     npe.getMessage());
```

```
    } catch (Exception e) {

      System.out.println("Client acceptance error  ==> " +
         e.getMessage() + e);

    } // end try - catch

    System.out.println("ConnectionLost status ==> " +
       connectionLost);
    System.out.println("Socket close? ==> " +
       clientSocket.isClosed());

    if (connectionLost == true) {

      server.setConnectionLost(true);

    } // end if

    dead = true;

  } // end run()

  /**
   * Method to assign parameters
   *
   * @param br - Input stream filters
   * @param bw - Output stream filters
   * @param buffer - array of characters
   * @param fileSize - the size of the file in bytes
   */
  public void setParameters(BufferedReader br,
       BufferedWriter bw, char[] buffer, int fileSize) {
    this.br = br;
    this.bw = bw;
    this.buffer = buffer;
    this.fileSize = fileSize;

  } // end setParameters()

  /**
   * Method to send the character to the client
   *
   * @param connectionLost - the status of the connection
   */
  public void setConnectionLost(boolean connectionLost) {
```

```
  this.connectionLost = connectionLost;

} // end setConnectionLost()

/**
 * Method to close the connection
 *
 * @throws IOException
 */
public void close() throws IOException {

  if (clientSocket.isClosed() == false) {

    clientSocket.close();

  } // end if

  if (br != null) {

    br.close();

  } // end if

  if (bw != null) {

    bw.close();

  } // end if

} // end close

/**
 * Method to assign parameters
 *
 * @param input - the request string from the client
 */
private void handle(String input) {

  // sending size
  if (input.equals("/size")) {

    String size = Integer.toString(buffer.length);
    System.out.println("size : " + size + "");
    send(size);

  } // end if
```

```
//sending data
else if (input.startsWith("/get ")) {

  System.out.println("Receive command -->" + input);
  StringTokenizer st = new StringTokenizer(input);
  String[] commandArray = new String[2];
  int counter = 0;

  // keep the indormation of the request in the array
  while (st.hasMoreTokens()) {

    commandArray[counter] = st.nextToken();
    counter++;

  } // end while

  if (commandArray[0].equalsIgnoreCase("/get")) {

    // convert to number of the index file to be read -
       -> can be started at 0 or at the byte it lost
    indexToReadFile =
       Integer.parseInt(commandArray[1]);

    server.h.setVisible(true);

    while (((indexToReadFile) < buffer.length) &&
       (connectionLost == false)) {

      try {

        // now array(file in bytes) has all bytes -->
           send each byte
        sendChar(buffer[indexToReadFile]);
        server.label2.setText("Character # " +
           indexToReadFile + " , " +
           buffer[indexToReadFile] + " is sent");

      } catch (Exception e) {

        System.out.println("While sending characters,
           Exception ==> " + e.getMessage());

      } // end try - catch

      indexToReadFile++; // increment the index
      System.out.println(indexToReadFile + ", " +
         buffer.length);
```

```
        if (connectionLost == true) {

          System.out.println("Next indexToReadFile ==> "
             + indexToReadFile);

          // break; //exit the while loop

        } // end if

      } // end while

      if (indexToReadFile == buffer.length) {

        JOptionPane.showMessageDialog(server, "End of
           File. Thread# " + ID + " done!");
        server.h.setVisible(false);

      } // end if

    } // end if

  } else if (input.startsWith("/Setup")) {

    System.out.println("Receive command -->" + input);
    StringTokenizer st = new StringTokenizer(input);
    String[] commandArray = new String[3];
    int counter = 0;

    // keep the indormation of the request in the array
    while (st.hasMoreTokens()) {

      commandArray[counter] = st.nextToken();
      System.out.println(commandArray[counter]);
      counter++;

    } // end while

    RTSPSeqNb = Integer.parseInt(commandArray[1]);
    RTP_dest_port = Integer.parseInt(commandArray[2]);

    // do setup session automatically
    // init the VideoStream object:
    try {

      video = new VideoStream(fileName);
```

```
} catch (Exception e) {

  System.out.println("problem creating video stream"
     + e);

} // end try - catch

// init RTP socket
try {

  RTPsocket = new DatagramSocket();

} catch (Exception e) {

  System.out.println("problem creating Datagram
      Socket" + e);

} // end try - catch

System.out.println("RTPsocket created!");

// Wait for the SETUP message from the client
int request_type;
done = false;

ClientIPAddr = clientSocket.getInetAddress();
System.out.println("ClientIPAddr = " + ClientIPAddr);

server.g.setVisible(true);

while (!done) {

  // parse the request
  request_type = parse_RTSP_request(); // blocking

  if ((request_type == PLAY)) {

    if (RTSPSeqNb == 0) {

      System.out.println("In else if 'PLAY' before
         start the timer");

      // start sending the video
      timer.start();

    } else {
```

```
// skip to the point it lost the connection
System.out.println("RTSP Sequence number = " +
   RTSPSeqNb);

// do the iteration, just reading - no sending
for (int i = 0; i < RTSPSeqNb; i++) {

  try {

    imagenb++;
    System.out.println("\nIn for loop, discard
       frame# " + i);

    // get next frame to send from the video,
       as well as its size
    int image_length = video.getnextframe(buf);
    System.out.println("image_length = " +
       image_length);

   // Builds an RTPpacket object containing the
      frame
    RTPpacket rtp_packet = new
       RTPpacket(MJPEG_TYPE, imagenb, imagenb *
       FRAME_PERIOD, buf,
        image_length);

    // get to total length of the full rtp
       packet to send
    int packet_length = rtp_packet.getlength();
    System.out.println("packet_length = " +
       packet_length);

    // retrieve the packet bitstream and store
       it in an array of bytes
    byte[] packet_bits = new
       byte[packet_length];
    rtp_packet.getpacket(packet_bits);

  } catch (Exception e) {

    System.out.println(e.getMessage());

  } // end try - catch

} // end for
```

```
          /** After reading the undesired VDO part, start
              to send the rest
           * of the video to the client */
          timer.start(); // start timer

        } // end if - else

      } else if (request_type == PAUSE) {

        System.out.println("In else if 'PAUSE'");
        timer.stop(); // stop timer

      } else if (request_type == CLOSE) {

        System.out.println("In else if 'CLOSE'");
        timer.stop();// stop timer

        done = true;
        //RTPsocket.close();
        server.g.setVisible(false);

      } // end if - else if

      try {

        // End of File
        if (imagenb >= VIDEO_LENGTH) {

          done = true;
          RTPsocket.close();

        } // end if

      } catch (Exception e) {

        System.out.println("problem creating Datagram
           Socket" + e);

      } // end try - catch

    } // end while

  } // end else if "/Setup'

} // end handle()

/**
```

```
 * Method to send the message to the client
 *
 * @param message - the string request
 */
private void send(String message) {

  try {

    bw.write(message);
    bw.newLine();
    bw.flush();

  } catch (IOException ioe) {

    server.addTextArea("While sending command,
       IOException ==> " + ioe.getMessage());
  } // end try - catch

} // end send()

/**
 * Method to send the character to the client
 *
 * @param ch - the string request
 */
private void sendChar(char ch) {

  try {

    bw.write(ch);
    bw.newLine();
    bw.flush();

  } catch (IOException ioe) {

    System.out.println(this + "While sending characters,
       IOException ==> " + ioe.getMessage());
    connectionLost = true;

  } catch (Exception e) {

    System.out.println(this + "While sending characters,
       Exception ==> " + e.getMessage());
    connectionLost = true;

  } // end try - catch
```

```
} // end sendChar()

/**
 * Method to parse the request from the client
 *
 * @return request_type
 */
private int parse_RTSP_request() {

  int request_type = -1;

  try {

    //parse request line and extract the request_type:
    String RequestLine = br.readLine();

    System.out.println("RTSP Server - Received from
       Client:");
    System.out.println("Received command --> " +
       RequestLine);

    StringTokenizer tokens = new
      StringTokenizer(RequestLine);
    String request_type_string = tokens.nextToken();

    if ((new
         String(request_type_string)).compareTo("PLAY")
         == 0) {
      request_type = PLAY;
      System.out.println("Request type --> " +
         request_type);

    } else if ((new
        String(request_type_string)).compareTo("PAUSE")
        == 0) {

      request_type = PAUSE;
      System.out.println("Request type --> " +
         request_type);

    } else if ((new
        String(request_type_string)).compareTo("CLOSE")
        == 0) {

      request_type = CLOSE;
      System.out.println("Request type --> " +
         request_type);
```

```
        } // end if - else if

      } catch (Exception ex) {

        System.out.println("Exception caught: " + ex);
        System.exit(0);

      } // end try - catch

      return (request_type);

    } // end parse_RTSP_request()

    /**
     * Method to send the response to the client
     */
    private void send_RTSP_response() {

        try {

            System.out.println("EOF sent!");
            // write the message to the server using buffer
               writer
            bw.write("EOF");
            bw.flush();

        } catch (Exception ex) {

            System.out.println("in send Exception caught: " +
               ex);
            System.exit(0);

        } // end try - catch

      } // send_RTSP_response()


    //-------------- I N N E R   C L A S S ---------------//

    //------------------------------------------------
    //
    //          Action Listener Methods:
    //
    //------------------------------------------------
```

```
private class TimerHandler extends JFrame implements
    ActionListener {

  //------------------------------------
  //        Handler for timer
  //------------------------------------

  public void actionPerformed(ActionEvent e) {

    System.out.println("\nIn the timer");

    // if the current image nb is less than the length of
       the video
    if (imagenb < VIDEO_LENGTH) {

      imagenb++;// update current imagenb

      try {

        // get next frame to send from the video, as well
           as its size
        int image_length = video.getnextframe(buf);
        System.out.println("image_length = " +
          image_length);

        // Builds an RTPpacket object containing the
           frame
        RTPpacket rtp_packet = new RTPpacket(MJPEG_TYPE,
           imagenb, imagenb * FRAME_PERIOD, buf,
           image_length);

        // get to total length of the full RTP packet to
           send
        int packet_length = rtp_packet.getlength();
        System.out.println("packet_length = " +
           packet_length);

        // retrieve the packet bitstream and store it in
           an array of bytes
        byte[] packet_bits = new byte[packet_length];
        rtp_packet.getpacket(packet_bits);

        // send the packet as a DatagramPacket over the
           UDP socket
        senddp = new DatagramPacket(packet_bits,
           packet_length, ClientIPAddr, RTP_dest_port);
```

```
        // test the connection by using get command
        if (connectionLost == false) {

          //Thread.sleep(500); // make it longer
          RTPsocket.send(senddp);

          // update GUI
          server.label.setText("Send frame # " + imagenb
             + "\n");
          System.out.println("Send frame # " + imagenb);

        } else {

          System.out.println("\nPhysical connection is
             lost for thread# ." + ID);
          System.out.println("\nThe timer will be stop
             and wait for another request from the
             client!\n");

          // stop the timer
          timer.stop();
          server.g.setVisible(false); // disable the
             visibility of the GUI

        } // end if - else

      } catch (Exception ex) {

        System.out.println("Exception in the timer");
        System.out.println("Exception caught: " + ex);

        System.exit(0);

      } // end try - catch

    } else {

      // if we have reached the end of the video file,
         stop the timer
      timer.stop();
      System.out.println("The end of the video!");

      try {

        send_RTSP_response();

        int reply = parse_RTSP_request();
```

```
          if (reply == CLOSE) {

              done = true;
              //RTPsocket.close();
              server.g.setVisible(false);

          } // end if

        } catch (Exception exp) {}

      } // end if - else

    } // end actionPerformed()

  } // end inner class timerListener

} // end class ServerThread
```

```
/**
 * Title: API Development for Persistent Data Sessions
   Support
 * Description: Streaming Video with RTSP and RTP
 * Compiler : JBuilder 9
 * Author CPT.Chayutra Pailom THA
 * Date : January 20, 2005
 */

import java.io.*;

public class VideoStream {

//-----------------------------------------
//
//            Data Members:
//
//-----------------------------------------

  /** Video file */
  FileInputStream fis;

  /** current frame nb */
  int frame_nb;

//-----------------------------------------
//
//             Constructor:
//
//-----------------------------------------

  public VideoStream(String filename) throws Exception{

    //init variables
    fis = new FileInputStream(filename);
    frame_nb = 0;

  } // end constructor

  /**
   * Get the next frame
   *
   * @param frame array of byte of the video
   *
   * @return the next frame as an array of byte and the
     size of the frame
   *
```

```
   * @throws Exception
   */
  public int getnextframe(byte[] frame) throws Exception {

    int length = 0;
    String length_string;
    byte[] frame_length = new byte[5];

    // Read current frame length
    fis.read(frame_length,0,5);

    // Transform frame_length to integer
    length_string = new String(frame_length);
    length = Integer.parseInt(length_string);

    return(fis.read(frame,0,length));

  } // end getnextframe()

} // end class VideoStream
```

APPENDIX B. CLASS DIAGRAMS

In this section, the class diagrams from Chapter IV are shown in order as following:

- class diagram of the application client
- Class diagram of the persistency API
- Class diagram of the RTP packet
- Class diagram of the application server
- Class diagram of the application proxy server
- Class diagram of the streaming video for RTP and RTSP

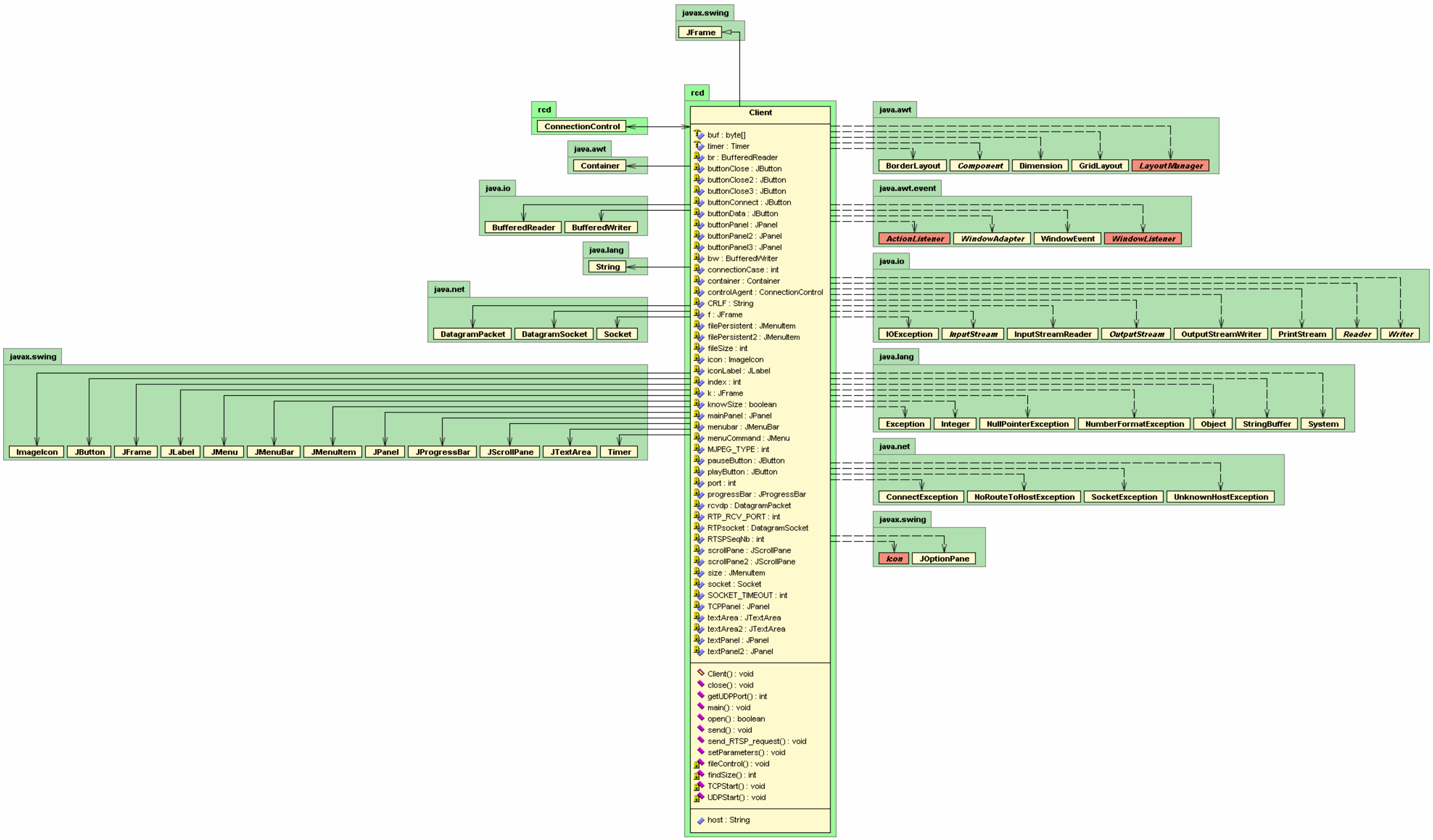

Figure 40. Class diagram of the application client

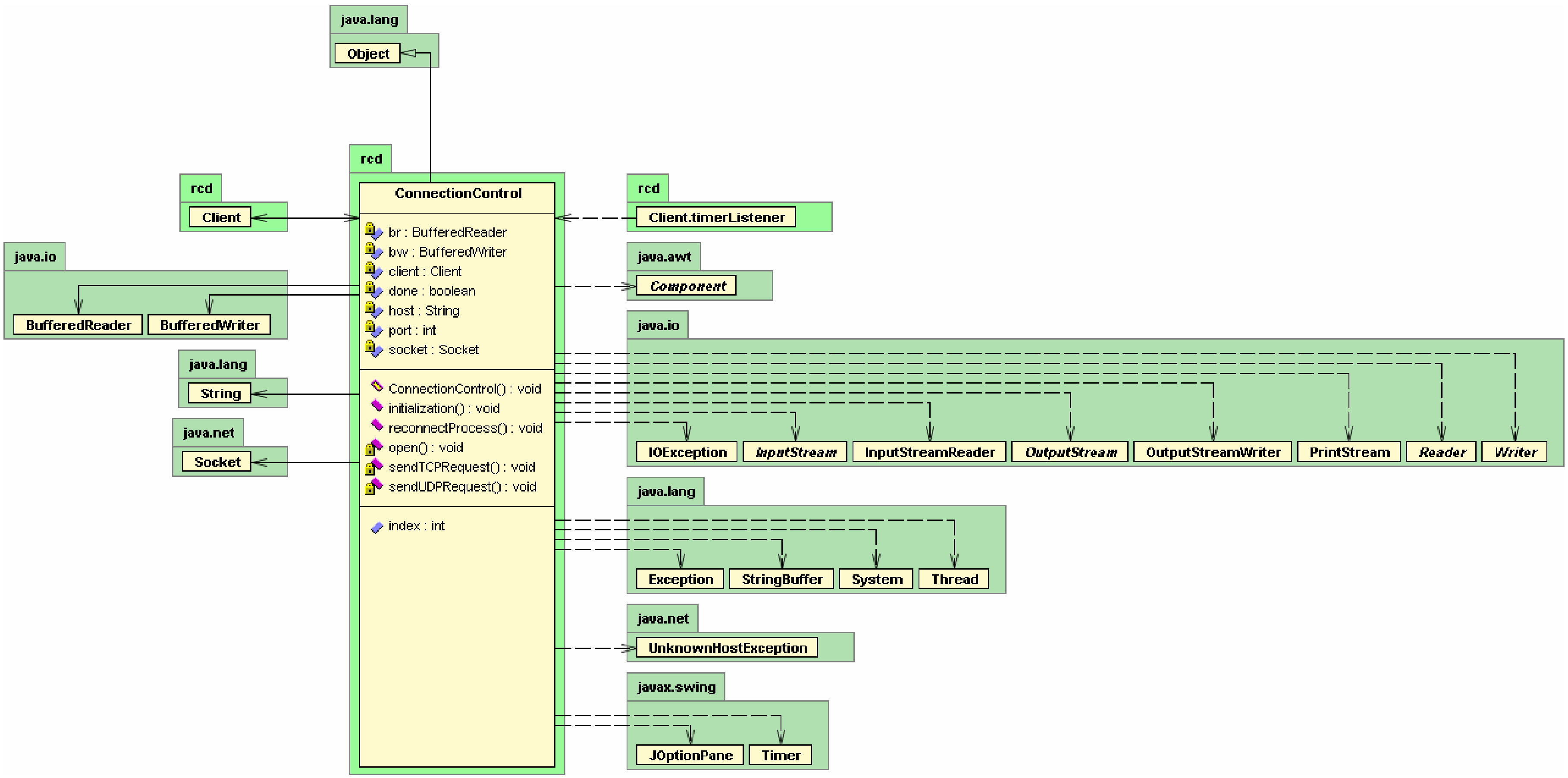

Figure 41. Class diagram of the persistency API

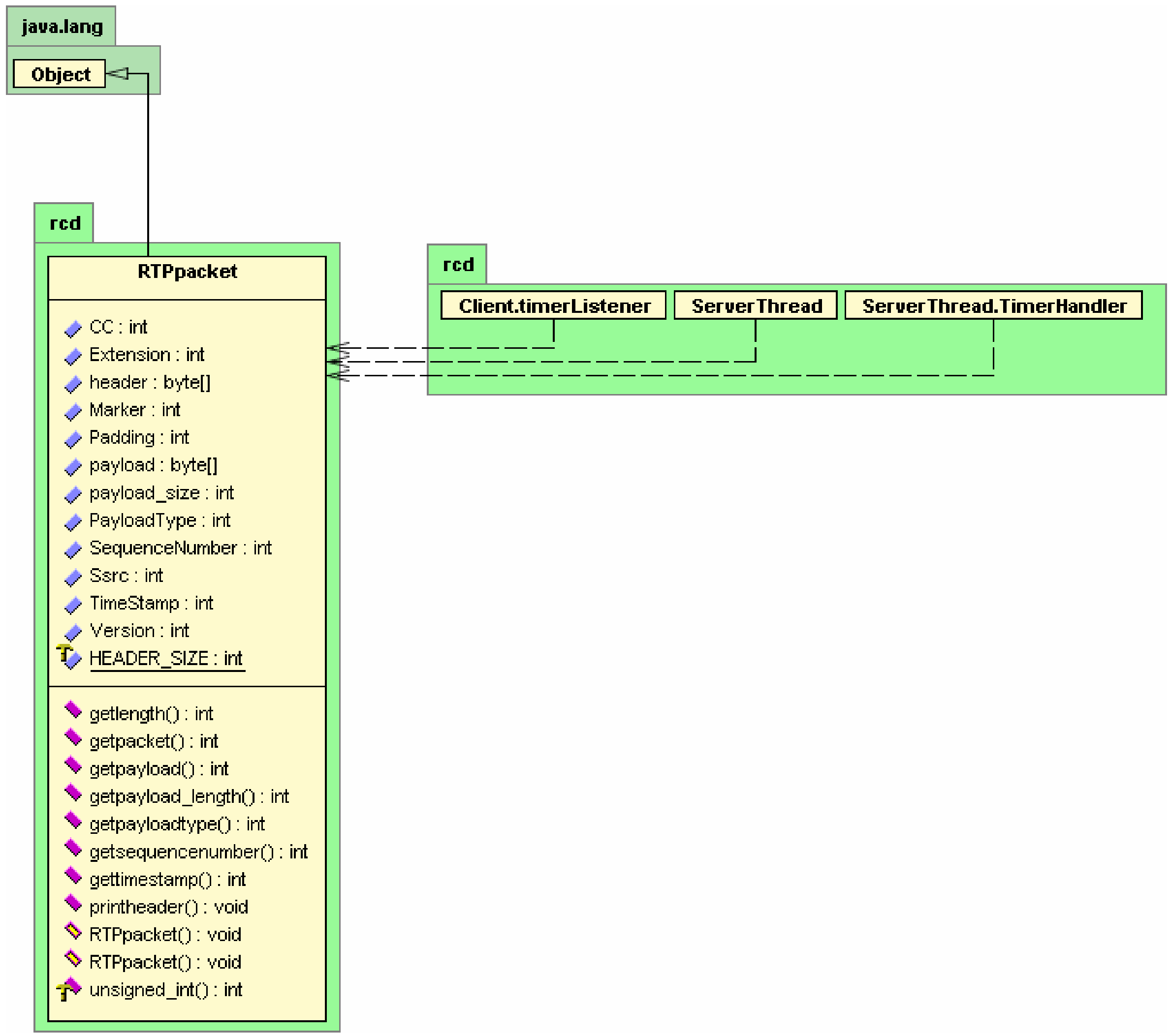

Figure 42. Class diagram of the RTP packet

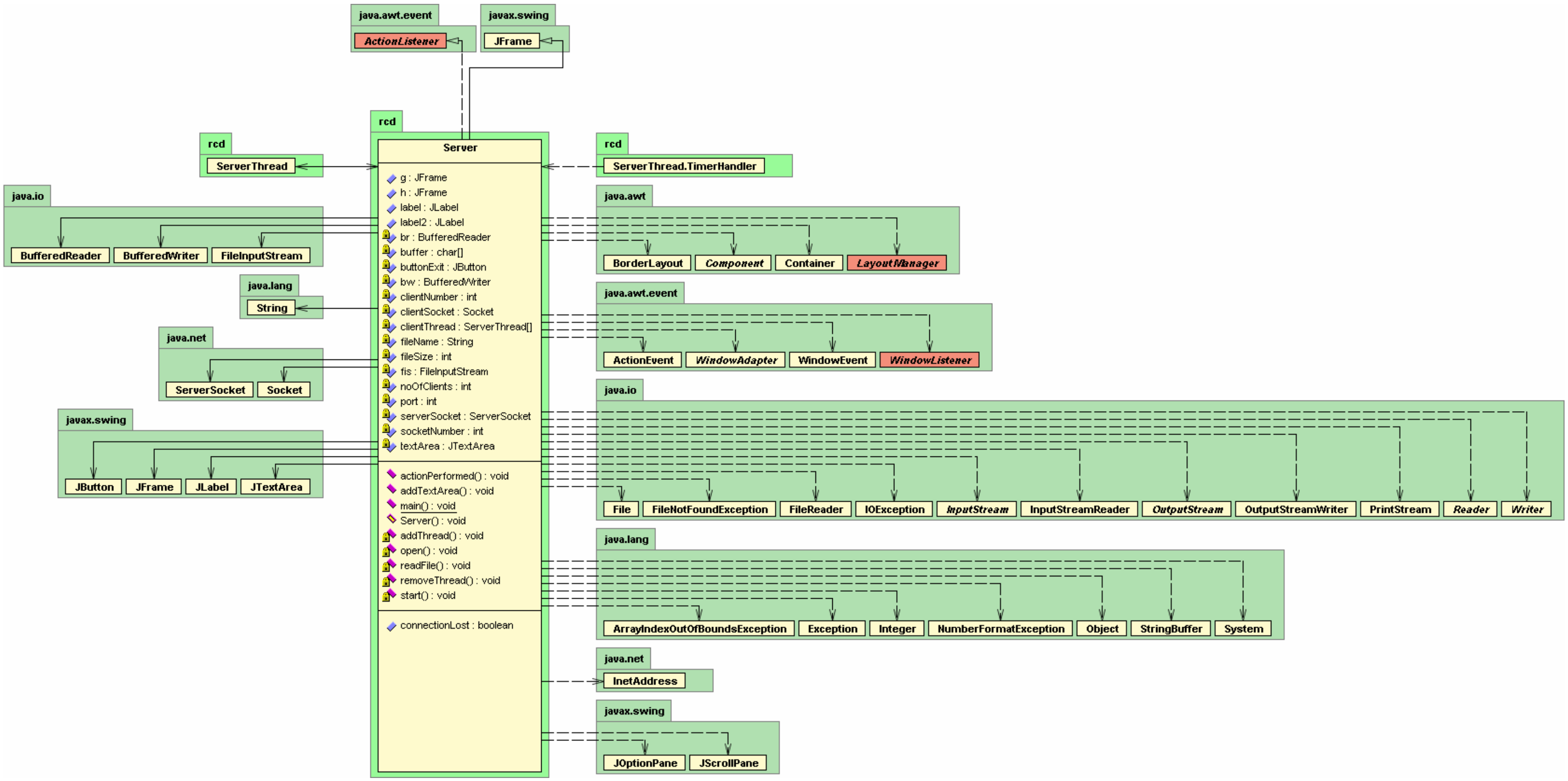

Figure 43. Class diagram of the application server

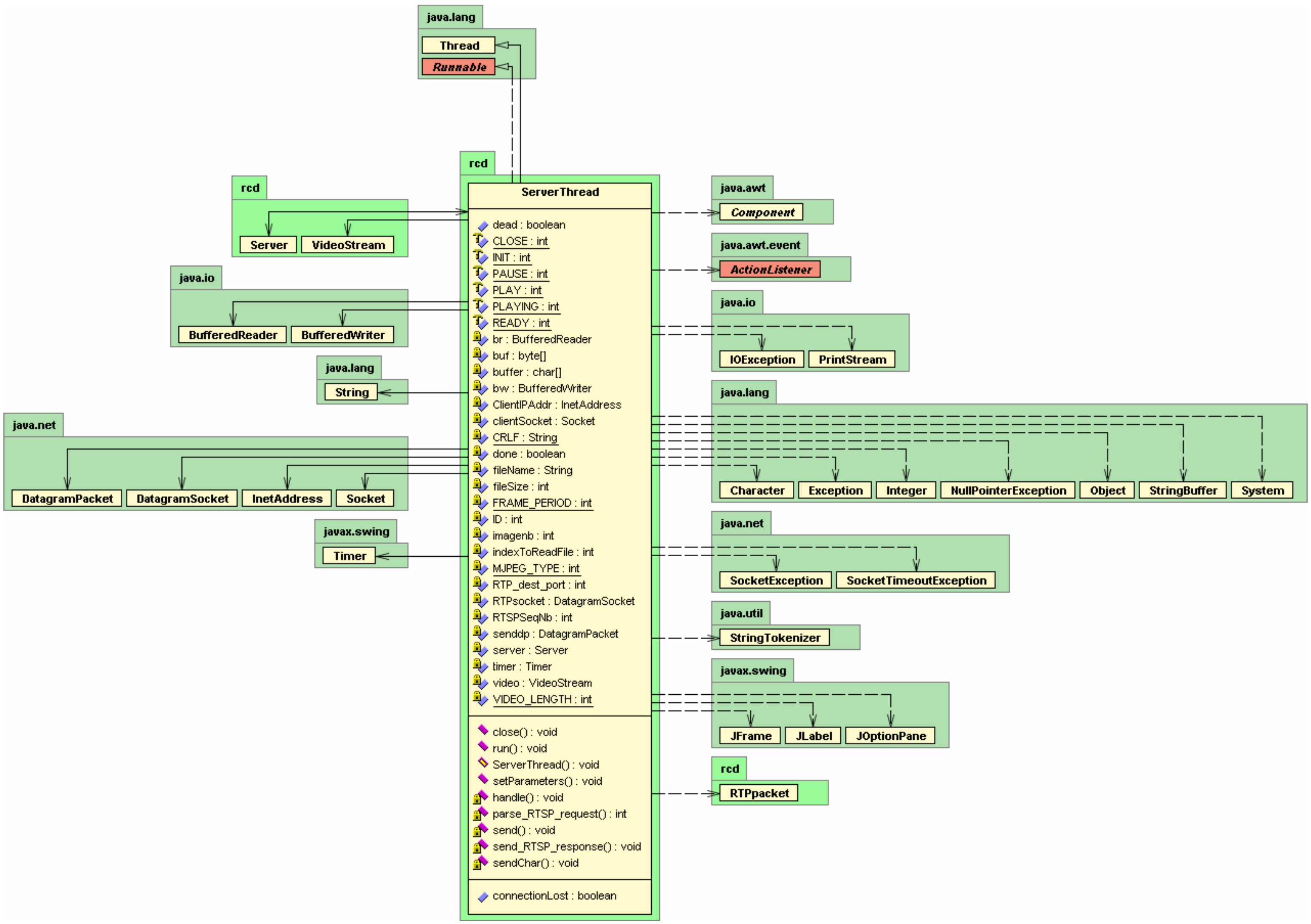

Figure 44. Class diagram of the application proxy server

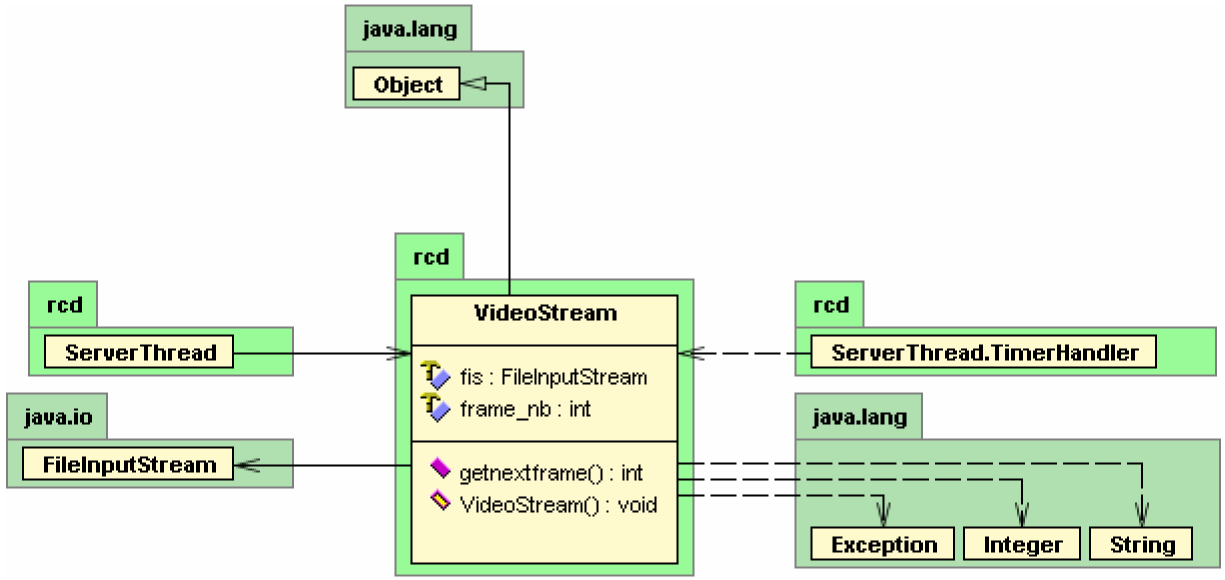

Figure 45. Class diagram of the streaming video for RTP and RTSP

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Wikipedia, "Session" The free encyclopedia. Internet Available http://en.wikipedia.org/wiki/Main Page (2 February 2005).

[2] Java, "Networking Features". Internet Available http://java.sun.com/j2se/1.5.0/docs/guide/net/ (5 February 2005).

[3] Disco Lab, "Laboratory for Network Centric Computing". Internet Available http://discolab.rutgers.edu/index.html (20 December 2004)

[4] Pantoleon, Perliklis K. "Reliable Content Delivery using Persistent Data Sessions in Highly Mobile Environment", Master Thesis, March 2004.

[5] Stevens, R. "UNIX Network Programming", Englewood Cliffs: Prentice-HalL Inc., 1990.

[6] Developer Works, "Developing an On Demand Workplace, Part 7: Really going mobile" IBM, Internet Available http://www-106.ibm.com/developerworks/library/i-workplace7/ (15 February 2005).

Kurose, James F. and Ross, Keith W. Computer networking, A Top-Down Approach Featuring the Internet Second Edition, Amsherst: University of Massachusetts and Eurecom Institute.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Peter J. Denning, Chairman, Ph.D, Code CS
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

4. Su Wen, Assistant Professor, Ph.D, Code CS
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

5. Arijit Das, Research Associate, Code CS
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

6. Captain Chayutra Pailom
   Royal Thai Supreme Command Headquarter
   Bangkok, Thailand